Leveraging Posit Arithmetic in Deep Neural Networks

Master's Thesis
Course 2020–2021

Author
Raul Murillo Montero

Advisors
Dr. Alberto A. del Barrio García
Dr. Guillermo Botella Juan

Master in Computer Science Engineering
Faculty of Computer Science
Complutense University of Madrid

This document is prepared for double-sided printing.

Leveraging Posit Arithmetic in Deep Neural Networks

Master's Thesis in Computer Science Engineering
Department of Computer Architecture and Automation

Author
Raul Murillo Montero

Advisors
Dr. Alberto A. del Barrio García
Dr. Guillermo Botella Juan

Call: *January 2021*
Grade: *10 - Matrícula de honor*

Master in Computer Science Engineering
Faculty of Computer Science
Complutense University of Madrid

Course 2020–2021

Acknowledgements

To my advisors, Alberto and Guillermo, thank you for giving me the opportunity to embark on this project, for all the corrections, the constant suggestions for improvement, and your support.

Thanks to my family and to all of you who have accompanied me on this journey and have made it a little easier.

COMPLUTENSE UNIVERSITY OF MADRID

Abstract

Faculty of Computer Science
Department of Computer Architecture and Automation

Master in Computer Science Engineering

Leveraging Posit Arithmetic in Deep Neural Networks

by Raul Murillo Montero

The IEEE 754 Standard for Floating-Point Arithmetic has been for decades implemented in the vast majority of modern computer systems to manipulate and compute real numbers. Recently, John L. Gustafson introduced a new data type called *posit*^{TM} to represent real numbers on computers. This emerging format was designed with the aim of replacing IEEE 754 floating-point numbers by providing certain advantages over them, such as a larger dynamic range, higher accuracy, bitwise identical results across systems, or simpler hardware, among others. The interesting properties of the posit format seem to be really useful under the scenario of deep neural networks.

In this Master's thesis, the properties of posit arithmetic are studied with the aim of leveraging them for the training and inference of deep neural networks. For this purpose, a framework for neural networks based on the posit format is developed. The results show that posits can achieve similar accuracy results as floating-point numbers with half of the bit width without modifications in the training and inference flows of deep neural networks. The hardware cost of the posit arithmetic units needed for operating with neural networks (this is, additions and multiplications) is also studied in this work, obtaining great improvements in terms of area and power savings with respect state-of-the-art implementations.

Keywords

Posit arithmetic, Deep neural networks, Training, Inference, Adder, Multiplier, Computer arithmetic

UNIVERSIDAD COMPLUTENSE DE MADRID

Resumen

Facultad de Informática
Departamento de Arquitectura de Computadores y Automática

Máster en Ingeniería Informática

Aprovechando la Aritmética Posit en las Redes Neuronales Profundas

por Raul Murillo Montero

El estándar IEEE 754 para aritmética de coma flotante se ha implementado durante décadas en la gran mayoría de los sistemas informáticos modernos para manipular y calcular números reales. Recientemente, John L. Gustafson introdujo un nuevo tipo de datos llamado *posit*^{TM} para representar números reales en computadores. Este formato emergente fue diseñado con el objetivo de reemplazar a los números de coma flotante IEEE 754 proporcionando ciertas ventajas sobre ellos, como un mayor rango dinámico, mayor precisión, resultados idénticos bit a bit en todos los sistemas o un hardware más simple, entre otras. Las interesantes propiedades del formato posit parecen ser realmente útiles en el escenario de redes neuronales profundas.

En este trabajo de fin de máster se estudian las propiedades de la aritmética posit con el fin de aprovecharlas para el entrenamiento e inferencia de redes neuronales profundas. Para ello, se desarrolla un framework para redes neuronales basadas en el formato posit. Los resultados muestran que los posits pueden lograr resultados de precisión similares a los números en coma flotante con la mitad de la anchura de bits sin modificaciones en los flujos de entrenamiento e inferencia de las redes neuronales profundas. En este trabajo también se estudia el coste hardware de las unidades aritméticas posit necesarias para operar con redes neuronales (es decir, sumas y multiplicaciones), obteniendo grandes mejoras en términos de área y ahorro de energía con respecto a las implementaciones del estado del arte.

Palabras clave

Aritmética posit, Redes neuronales profundas, Entrenamiento, Inferencia, Sumador, Multiplicador, Aritmética de computadores

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Motivation

Throughout the history of mankind, there have been multiple numbering systems, each with its own advantages and disadvantages. The Roman numerals, used for more than twenty centuries throughout hole Europe, are a base-10 system that represents numbers by combinations of letters from the Latin alphabet. This format, even providing a compact representation in some cases, is not easy to compute on, and non-trivial calculations are usually performed using some mechanical device such as an abacus. Probably due to this, Roman numerals were replaced by Arabic numerals from the 15th century, relegating the former to aesthetic uses.

The Arabic numerals improved human interpretation and computation, but as they are also a base-10 system, some problems arise when trying to adopt this system in computers that use a binary system, that is, that use only two symbols: typically "0" (zero) and "1" (one). As can be seen, building a good number system is not a straightforward task.

Representing real numbers is even more challenging, especially in computers or any device that can handle only discrete and finite information. In contrast with integer numbers, which are a discrete set and so relatively easy to map a subset to such a device, reals are a continuous set, and therefore no injective mapping exists to represent a subset of them. The IEEE Standard for Floating-Point Arithmetic (IEEE 754) [1], [2] is the most common implementation that modern computing systems have adopted. It defines several sizes and specifications of floating-point numbers. However, multiple deficiencies in the IEEE 754 standard have been identified since its adoption in 1985, such as subtractive cancellation, redundancy of NaNs, signed zeros or inconsistency of results across machines due to the different rounding methods [3]. To address these shortcomings, in 2017 a new data type called a posit was proposed as a direct drop-in replacement for IEEE 754 floating-point numbers [4].

Originally, floating-point arithmetic was intended for scientific applications, but over the years, more and more applications require this type of computation, and the vast majority of modern computers have a floating-point coprocessor. This has also allowed the software to evolve, being able to be increasingly complex. Machine learning, and in particular deep learning, are a set of Artificial Intelligence (AI) techniques and algorithms that, in the last decade, have revolutionized several fields, from computer vision, speech and language understanding, up to medicine, automotive industry, and finance. For this reason, together with the promise to help in the future challenges of our society, deep learning is taking on an increasingly important role. The exceptional performance that this technology has achieved in recent years has been possible, mainly due to the large increase in available datasets and the computational resources to analyze them [5]. However, this trend of increasing deep learning models runs up against the end of Moore's law and Dennard scaling

[6]. The estimate of Gordon Moore in the mid-1960s began to fail at the turn of the century, and in recent years the gap between the number of transistors on a real chip and that predicted by Moore has grown. In a similar manner, Dennard scaling began to slow significantly in 2007 and faded to almost nothing by 2012.

It turns out that maintaining performance improvement nowadays to enable new software capabilities, such as deep learning, is as important as challenging. An interesting research direction in computer architecture is the use of Domain-Specific Architectures (DSAs), a class of processors tailored for a specific domain or class of applications. That is the case of Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs) or even more specific Google's Tensor Processing Units (TPUs) [7]. DSAs (also called accelerators) are closely tailored to the needs of a certain application and, thus, can achieve higher performance and greater energy efficiency than general-purpose CPUs. With respect to machine learning applications, several recent works have demonstrated that the IEEE floating-point format is quite inefficient for these tasks. As a consequence, both Google and Microsoft have developed their own alternative formats to IEEE floating point for their AI services, and more efficient formats are required for these king of applications to run on mobile, Internet of Things (IoT) or embedded devices. The interesting properties of posits and their recent breakthroughs in this area make this format a suitable alternative to the IEEE 754 standard for deep learning. A simple change to a new number system might improve the scale and cost of these applications by orders of magnitude, contributing this way to the machine learning revolution.

1.2 Objectives

The principal purpose of this Master's thesis is to determine if posit arithmetic could serve as a suitable replacement for the current IEEE 754 floating-point format in deep learning applications. Such a goal might seem quite complex, so in this case it will be interesting to apply the well-known divide-and-conquer strategy. Thus, this work has the following specific objectives:

- To acquire a general knowledge of floating-point arithmetic.
- To understand the design of posit arithmetic, how it works and how it differs from the floating-point standard.
- To explore the use of posit arithmetic in deep neural networks in both inference and training stages, its benefits and drawbacks compared to the widely used floating-point format.
- To design the minimum posit operators required to perform inference on deep neural networks.

1.3 Document overview

This document is a faithful reflection of the research process carried out from the very beginning. The rest of the document is structured as follows. Chapter 2 introduces the necessary notions to understand this MSc thesis. It explains with detail some basic concepts of computer arithmetic for real numbers, the posit format, its properties and drawbacks. In Chapter 3 the focus is set to deep neural networks. After a brief revision of concepts, it explains how posit arithmetic can be used, so that its benefits can be exploited in this area. Chapter 4 deals with hardware design

and implementation. Posit arithmetic units for addition/subtraction and multiplication are proposed and compared with previous works. Finally, in Chapter 5 the conclusions of this work are summarized and future lines of work are proposed.

1.4 Summary of contributions

This Master's thesis shows several contributions to the development of posit arithmetic and its application to deep learning.

First, posits were recently proposed (2017), and not many software tools that implement this datatype are available. Such tools facilitate learning and research, so they are essential for the development of this new format. In particular, there are no previous open-source deep learning libraries based on posits, and the work in this MSc thesis tries to fill this gap, as described in Chapter 3. As a result of this work, an open-source posit-based framework for deep neural networks called Deep PeNSieve has been implemented. Such a framework, licensed under the Apache License 2.0, is included in the official Unum & Posit page as part of the software development efforts[1].

The development of posit arithmetic units is also in an early stage, with some functional operators been presented so far, but still far away from floating-point units, which are quite optimized through decades of research. In this line of research and to contribute to the open-source hardware trend, the design of addition and multiplication posit units detailed in Chapter 4 are integrated into FloPoCo, an open-source command-line tool for generating floating-point (and now also posit) cores for FPGAs (all rights reserved). In addition, FloPoCo generated instances are publicly available under the GPL v3.0 license to facilitate its use and dissemination. Experiments show an improvement in terms of area, power and energy with respect to state-of-the-art works.

Below is the list of publications, arranged according to the order of appearance in this MSc thesis:

- Chapter 3: *Evaluation of Posit Arithmetic in Deep Neural Networks*
  - R. Murillo, A. A. Del Barrio, and G. Botella, "Deep PeNSieve: A deep learning framework based on the posit number system", *Digital Signal Processing: A Review Journal*, vol. 102, p. 102 762, 2020. DOI: 10.1016/j.dsp.2020.102762
  - R. Murillo Montero, A. A. Del Barrio, and G. Botella, "Template-based posit multiplication for training and inferring in neural networks", *arXiv e-prints*, 2019. arXiv: 1907.04091
- Chapter 4: *Design and Implementation of Posit Arithmetic Units*
  - R. Murillo, A. A. Del Barrio, and G. Botella, "Customized Posit Adders and Multipliers using the FloPoCo Core Generator", in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2020, pp. 1–5. DOI: 10.1109/iscas45731.2020.9180771
- Other publications that are not included in this MSc thesis:
  - R. Murillo, A. A. Del Barrio, and G. Botella, "La aritmética del futuro: Una reflexión sobre los planes de estudio", *Enseñanza y Aprendizaje de Ingeniería de Computadores*, vol. 10, 2020. DOI: 10.30827/Digibug.64781

[1] posithub.org/docs/PDS/PositEffortsSurvey.html, as of January 10, 2021.

2 | Posit Arithmetic

Historically, Computer Arithmetic has been an essential branch of Computer Science in general and of Computer Architecture in particular. However, in the last decade, with the explosion of machine learning techniques and especially Neural Networks (NNs), new arithmetic formats have entered the scene. In this chapter, the most significant of these are reviewed, especially the posits, introduced in 2017 by John L. Gustafson as a direct drop-in replacement for the IEEE 754 floating-point numbers.

One of the claims of posit arithmetic is that they provide higher accuracy than IEEE Standard 754 floating-point numbers (floats). Consider for example the following 2×2 linear system of equations.

$$\begin{pmatrix} 0.25510582 & 0.52746197 \\ 0.80143857 & 1.65707065 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0.79981812 \\ 2.51270273 \end{pmatrix} \tag{2.1}$$

One can easily check that the answer of system (2.1) is $x = -1$, $y = 2$. Since the system is 2×2, the Cramer's rule is a simple method for computing the solution. In order to avoid rounding errors when converting the decimal numbers to binary representation, system (2.1) can be re-written as (2.2).

$$\begin{pmatrix} 25510582 & 52746197 \\ 80143857 & 165707065 \end{pmatrix} \frac{1}{2^8} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 79981812 \\ 251270273 \end{pmatrix} \frac{1}{2^8} \tag{2.2}$$

The new system values can be accurately represented with double-precision IEEE floats (64 bits), and the solution remains the same. Nevertheless, when performing Cramer's rule with double-precision IEEE floats the obtained result is $x = 0$, $y = 2$. This terrible rounding error is produced due to an underflow on the x numerator expression. It is necessary to use quadruple-precision IEEE floats (128 bits) for obtaining the exact answer. On the other hand, the same computation using 64-bit posits, with 3 bits for the exponent, yields the correct solution. As can be seen, in certain circumstances, high precision floats may be safely replaced by lower precision posits.

Section 2.1 revises some basic concepts from computer arithmetic, with an emphasis on the floating-point format and the IEEE 754 standard. Section 2.2 introduces the preliminary idea of universal numbers that preceded the posit format. Section 2.3 describes in detail how this novel format encodes real values. Section 2.4 presents some of the most interesting and useful properties of this novel format, while its drawbacks are discussed in Section 2.5.

2.1 Preliminaries in computer arithmetic

Computer arithmetic is a field of computer science that investigates how computers should represent numbers and perform operations on them. All computer hardware, and practically all software, performs arithmetic by representing every number as a fixed-length sequence of 1s and 0s, or bits. However, there are a few differences when dealing with integers or real values in computers. Integers are often represented as a single sequence of bits, each representing a different power of two, with a single bit indicating the sign. Under this representation, integer arithmetic operates according to the "normal" (or symbolic) rules of arithmetic. On the other hand, there are multiple arithmetic formats for encoding real numbers in computers, and this is the case in which this Master's thesis will focus.

In scientific computing, most operations are on real numbers. There are two classical approaches to store real numbers (i.e., numbers with fractional component) in modern computing. These are fixed-point arithmetic and floating-point arithmetic. While the former of these formats can run on devices without the need for specific hardware for decimal arithmetic, the latter has greater precision and computational speed. In fact, floating-point arithmetic is ubiquitous in modern computing systems, and the preferred way computers approximate real numbers.

2.1.1 Floating-point arithmetic

Floating-point numbers (often called *floats*), as in scientific notations, are represented using an exponent (normally in base two) and a significand, except that this significand has to fit on a certain amount of bits.

The representation of a float is similar in concept to scientific notation, and consists of

- A signed number, referred to as the *significand*, *mantissa*, *coefficient*, or ambiguously *fraction*. This number is encoded as a digit string of a given length.
- A signed integer exponent (normally in base two), which modifies the magnitude of the number.

To obtain the value of the floating-point number F, the significand or mantissa M is multiplied by the base β raised to the power of the exponent E, as indicated by Equation (2.3).

$$F = M \times \beta^E \tag{2.3}$$

The way in which the significand (including its sign) and exponent are stored in a computer is implementation-dependent, but it is quite common that the first bit represents the sign, the next bits the exponent, and finally the significand.

The term *floating point* refers to the fact that a number's radix point can "float"; that is, it can be placed anywhere relative to the significant digits of the number, in a similar way as in common scientific notation. A floating-point system can be used to represent, with a fixed number of digits, numbers of different orders of magnitude, so it can be often found in systems which include very small and very large real numbers. However, since it uses only a finite number of bits, not all numbers can be represented under this format. Over the years, a variety of floating-point representations have been used in computers, but in 1985, the IEEE 754 Standard for Floating-Point Arithmetic was established. Since then, the most commonly encountered representations are those defined by the IEEE.

2.1.2 The IEEE 754 standard for floating-point arithmetic

Some decades ago, issues like the length of the mantissa and the rounding behavior of operations for floating-point numbers could differ between computer manufacturers, and even between models from one manufacturer. Obviously, this led to problems of portability and reproducibility of results. The IEEE Standard for Floating-Point Arithmetic (IEEE 754) solved these shortcomings by providing definitions for arithmetic formats, rounding schemes, operations, representation of special numbers, and exception handling [1].

The original IEEE 754-1985, which is implemented in the vast majority of modern computers, defines two basic formats, namely single (sometimes called float32 or FP32) and double. Note that this nomenclature is frequently used in basic data types from programming languages like C, C++ or Java. Representations of numbers in these formats are encoded in k bits in the following three fields, as shown in Figure 2.1:

- 1-bit sign S
- w-bit biased exponent $E = e + bias$
- $(t = p - 1)$-bit trailing significand field digit string $T = d_1 d_2 \dots d_{p-1}$; the leading bit of the significand, d_0, is implicitly encoded in the biased exponent E.

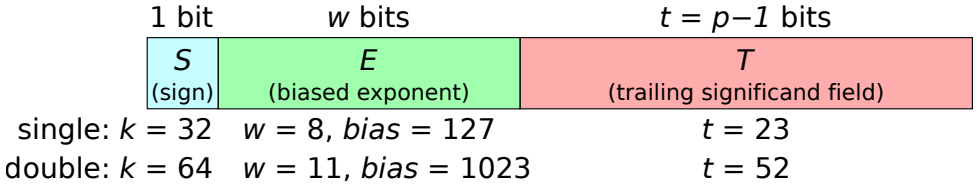


FIGURE 2.1: Encoding of the binary floating-point formats

The value v of a floating-point number is inferred from the aforementioned fields as follows:

a) If $E = 2^w - 1$ and $T \neq 0$, then v is Not-a-Number (NaN), and d_1 shall distinguish between qNaN (quiet) and sNaN (signaling).

b) If $E = 2^w - 1$ and $T = 0$, then $v = (-1)^S \times (+\infty)$.

c) If $1 \leq E \leq 2^w - 2$, the value of the corresponding floating-point number is $v = (-1)^S \times 2^{E-bias} \times (1 + 2^{1-p} \times T)$; thus normal numbers have an implicit leading significand bit of 1.

d) If $E = 0$ and $T \neq 0$, the value of the corresponding floating-point number is $v = (-1)^S \times 2^{emin} \times (0 + 2^{1-p} \times T)$, where $emin = 1 - bias$; this kind of numbers are called subnormal or denormalized, and have an implicit leading significand bit of 0.

e) If $E = 0$ and $T = 0$, then $v = (-1)^S \times (+0)$ (that is, signed zero).

Note that this format represents signed infinity and zeros, and multiple NaN exceptions. Denormalized numbers are non-zero numbers with magnitude smaller than the smallest normal number, and provide a gradual underflow, filling this gap around zero in floating-point arithmetic.

The IEEE 754 original standard from 1985 has two revisions, one in 2008, which extended the previous version with decimal floating-point arithmetic (radix 10), and another in 2019, which is a minor revision of IEEE 754-2008.

2.1.3 Other floating-point formats

In addition to the floating-point formats defined in the IEEE 754-1985, there are alternatives formats for floating-point computation. The revision of the standard (IEEE 754-2008), which renamed the previous single and double formats as binary32 and binary64 (indicating the number of occupied bits in their names), included definitions for binary16 (also called half precision, float16 or FP16) and binary128 (or quadruple precision). While the latter was designed for applications requiring results in higher than double precision, the former is intended for storage of floating-point values in applications where higher precision is not essential, such as machine learning. Thus, although implementations of the IEEE half-precision floating-point are relatively new, another 16-bit floating-point formats have been designed. That is the case of bfloat16 (Brain Floating Point), designed by Google [8], which also occupies 16 bits in computer memory and is widely used for machine learning and NNs. As depicted in Figure 2.2, this format is not so similar to binary16, but is a truncated (16-bit) version of the 32-bit IEEE 754 single-precision floating-point format.

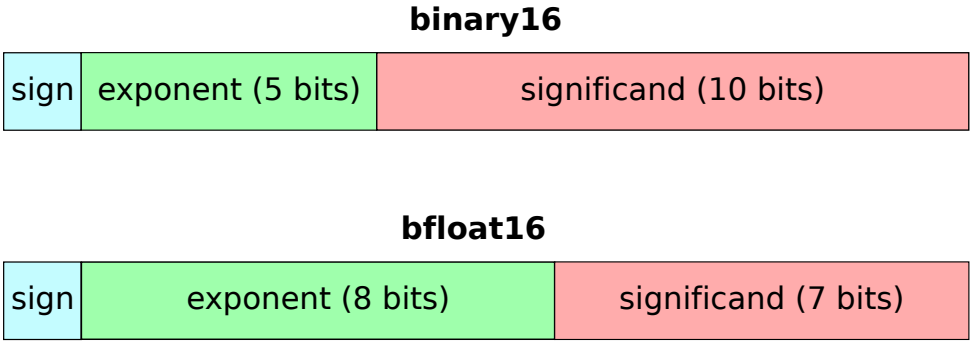


FIGURE 2.2: Layout of IEEE 754 half-precision and bfloat16 floating-point formats

Other alternatives to IEEE 754-2008 compliant arithmetic developed in the last years are Flexpoint, developed by Intel, High-Precision Anchored (HPA) numbers, developed by ARM, NVIDIA TensorFloat, Microsoft Floating Point (MSFP), and posit arithmetic, which is the focus of this MSc thesis. As can be seen, there is a trend for leading companies in the sector to develop their own arithmetic formats, most of them for use in deep learning applications, therefore replacing the floating-point standard used until now.

2.2 Background: Type I and Type II unums

The concept of *unum* (*u*niversal *num*ber) was proposed by John L. Gustafson in [9] as an alternative to the IEEE 754 arithmetic standard. This arithmetic format has several forms.

The original "Type I" unum is a superset of IEEE 754 Standard floating-point format. It uses a "ubit" at the end of the fraction to indicate whether the number corresponds to an exact value or lies in an open interval. Type I unum takes definition of the sign, exponent and fraction bit fields from IEEE 754, but the last two have variable-width. The bit lengths of such fields are indicated after the ubit. Although this format provides a simpler way to express interval arithmetic, its variable length demands extra management.

The "Type II" unum [10] was designed to resolve some of the shortcomings that Type I had, presenting a mathematical design based on the projective reals, but abandoning compatibility with the IEEE 754. This version has many ideal mathematical properties, such as symmetrical distribution for negative and inverse values, but relies on look-up tables for most operations. This limits the scalability to about 20 bits or less. Furthermore, fused operations such as dot product, which is used in a vast amount of applications, is quite expensive in this format. These drawbacks served as motivation of a search for a new format that would keep many of the Type II unum properties, but also be "hardware-friendly", which means, easily implementable on hardware.

2.3 The posit format

The "Type III" unum, better known as posit, as introduced by John L. Gustafson in 2017 "as a direct drop-in replacement for IEEE 754 standard for floating-point numbers" [4]. Posit numbers take the concept from the previous unum type while relaxing the perfect symmetry condition in order to perform hardware computations with similar logic to the existing floating-point format. In the case of posits, there are no open intervals, but all bit patterns represent different real values. The alternative to posits for interval arithmetic are called *valids*, but these latter are out of the scope of this work.

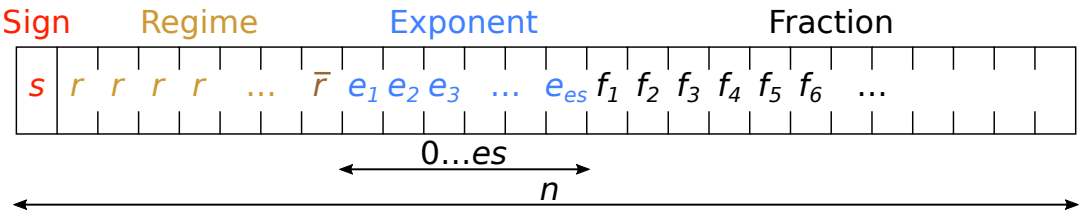


FIGURE 2.3: Layout of an Posit$\langle n, es \rangle$ number

A posit format is defined as a tuple $\langle n, es \rangle$, where n is the total bitwidth and es is the maximum number of bits reserved for the exponent field. As Figure 2.3 shows, a Posit$\langle n, es \rangle$ is encoded with four fields:

- **Sign.** As for floats or signed integers, the first bit stands for the sign: 0 for positive numbers, 1 for negative numbers. In the negative case, the 2's complement of the remaining bits must be taken to extract the following fields correctly.
- **Regime.** This field is unique to this number format. The regime consists of a sequence of m identical bits r terminated either with the negation of such value ($1 - r = \bar{r}$), as depicted in Figure 2.3, or with the least significant bit of the posit. This sequence encodes the scaling factor k, given by conditional Equation (2.4). For example, when the regime is 4-bits, pattern `1110` encodes $k = 2$, while `0001` stands for $k = -3$.

$$k = \begin{cases} -m & \text{if } r = 0 \\ m - 1 & \text{if } r = 1 \end{cases} \tag{2.4}$$

- **Exponent.** The following es bits encode another scaling factor e. Unlike with floats, the exponent is unbiased. As the length of the regime field is variable, there exists the possibility that some exponent bits (or even all of them) are shifted out of the bit-string. In such case, missing bits are considered as `0`.

- **Fraction.** The remaining bits after the exponent correspond to the fraction field, and they represent the fraction value f. This is similar as in the floats case, whit the only difference that the hidden bit is always 1, so denormalized numbers do not exist in the posit format. The value f is obtained from the unsigned integer represented by this field, so that $0 \leq f < 1$.

There are only two posit exception values: 0 (all 0 bits) and $\pm\infty$ (1 followed by all 0 bits). For the rest of bit strings, the real value X of a generic Posit$\langle n, es\rangle$ is expressed by Equation (2.5),

$$X = (-1)^s \times useed^k \times 2^e \times (1 + f), \tag{2.5}$$

where $useed = 2^{2^{es}}$.

The main differences with the IEEE standard for floating-point numbers are the utilization of an unsigned and unbiased exponent, if there exists such exponent field, the fraction hidden bit which is always 1, so there are no denormalized numbers in posit arithmetic, and the existence of the regime field. This variable size field allows to have more fraction bits for small values, increasing the precision at that range, while increasing the exponent for larger quantities, at the cost of loosing some precision. As can be seen in Table 2.1, posits with similar precision (number of sigificand bits) at 1 as IEEE floats provide a larger dynamic range[1]. It is noteworthy that, while the new regime field provides important scaling capabilities that improve the dynamic range of posits, detecting the resulting varying-sized fields adds an extra hardware overhead.

TABLE 2.1: Float and posit dynamic ranges for the same number of bits

	IEEE Float			Posit		
Bits	Exp. size	Dynamic Range	Precision	es Value	Dynamic Range	Precision
16	5	12.04	10	1	16.86	12
32	8	83.39	23	3	144.49	26
64	11	631.56	52	5	1194.49	56
128	15	9897.26	112	8	19420.05	117

2.4 Properties of the posit number system

The Posit Number System (PNS) provides compelling advantages over IEEE 754 floating-point numbers. Such advantages range from a more elegant design (in a mathematical sense) to mechanisms that reduce the rounding error in calculations. Below, the properties that make posits a great alternative to floats are presented.

2.4.1 Zero, infinite and NaN

As can be seen, posit numbers are coded in a similar manner as floats are, that is, a sign bit followed by the exponent of a scaling factor and a few fraction bits at the end. However, posit format allows a single representation for zero value, with all bits set to 0, in contrast with floating-point format, which presents positive and negative

[1]The dynamic range of a number system is the log base 10 of the ratio of the largest to smallest representable numbers.

zero values, depending on the sign bit. The corresponding bit pattern for negative zero in floating point, that is, a 1 bit preceded by all 0 bits, is used in posit format to represent the exception value $\pm\infty$, also called Not-a-Real (NaR). Floats present two different representations for positive and negative infinities, again according to the sign bit. In addition, the IEEE 754 format provides Not-a-Number (NaN) exceptions, which are not the same as infinity, although both are typically handled as special cases. In the IEEE floating-point standard, bit strings with all the exponent bits set to 1 represent a NaN. Just the first fraction bit is used to determine the type of NaN, while the rest of bits are often ignored in applications. Posit format solves this shortcoming by a single exception value. This allows not only to represent more values with the same number of bits, but to tremendously simplify the exception handling when designing posit units. This value is returned as a result of any invalid operation such as a zero division or the square root of a negative number. In fact, in the posit format there is no need for infinite values to handle arithmetic overflow since this format does not overflow or underflow, in such corner cases it just returns the maximum or minimum representable value, respectively.

2.4.2 Visualizing posits as projective reals

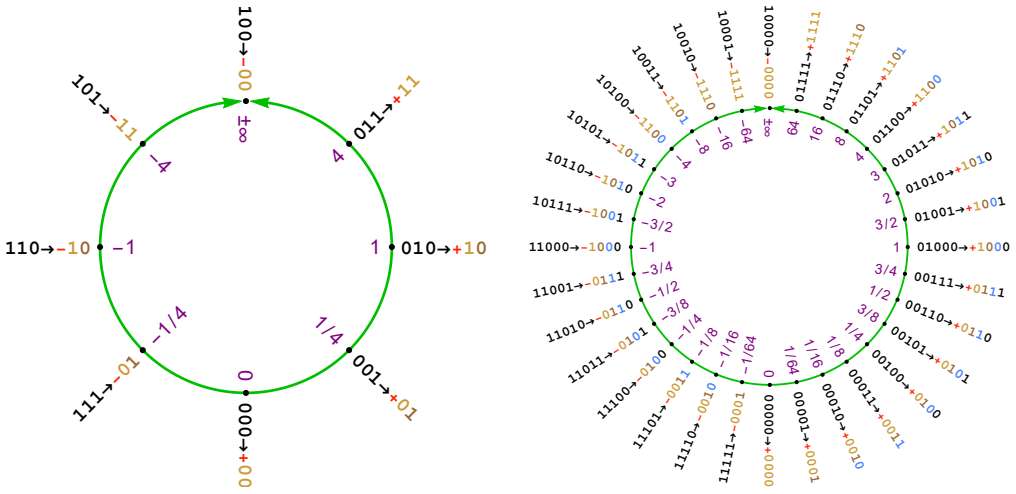
Posits were created as a modification of Type II unum to be more similar to the floating-point numbers. Therefore, posit format inherits one of the concepts of the previous unum type, the design based on the projective reals. Posits can be mapped into the real projective line[2] as depicted in Figure 2.4, which helps visualizing and understanding this novel format. Apart from the elegant mathematical design, these geometric representations reveal some of the properties of the posit format.

- As mentioned earlier, and in contrast with the IEEE 754 standard, posits have a single representation of the 0 and NaR values, and the latter is the only exception value used in this format.
- Posit numbers with their corresponding bit strings form a totally ordered set. As there are no two posit numbers with the same bit representation, it is a strict total order relation. Thanks to this property, posit comparison can be done as a signed integers operation. Again, this is not possible in the case of floats, where different bit string may represent the same real value ($0 = -0$) and negative numbers are not encoded in 2's complement.
- For every posit configuration, about half of the representable values concentrate in the interval $[-1, 1]$. This makes posit values to follow a normal distribution centered at 0, the so-called tapered precision. This is shown in Figure 2.5. For this reason, posits present a higher accuracy in that interval, where take place most of the computations from many different areas, such as deep learning, digital signal processing or numerical analysis.

2.4.3 Overflow, underflow and rounding

As mentioned earlier, in posit arithmetic there are no overflow to NaR and underflow to 0 phenomena. When the result of an operation is greater (respectively lower) than the maximum (respectively minimum) representable posit number, the result is rounded to that extreme value. For all other cases where a real number is not exactly

[2] In geometry, a real projective line is an extension of the usual concept of line that incorporates a point at infinity; i.e., the one-point compactification of $\mathbb{R}$.



(A) Posit⟨3, 1⟩ (B) Posit⟨5, 1⟩

FIGURE 2.4: Visual representation in the real projective line of different posit formats

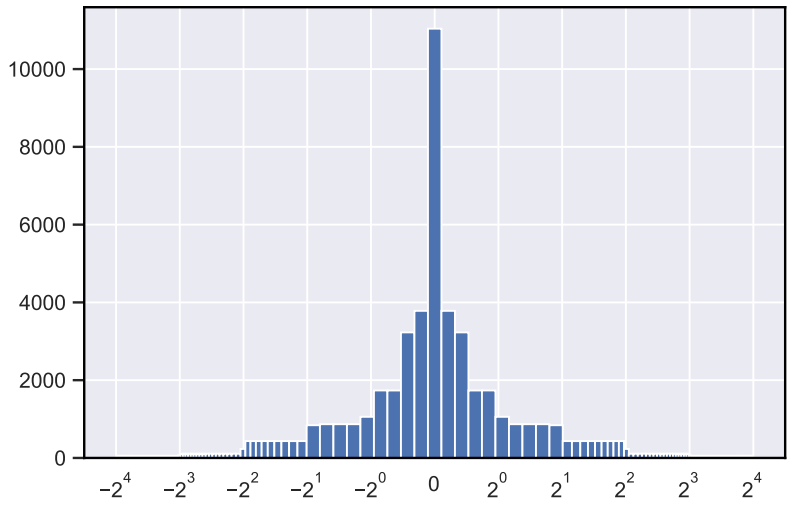


FIGURE 2.5: Histogram of representable values for Posit⟨16, 1⟩

representable in a posit format, the standard specifies the use of the *round-to-nearest-even* scheme, i.e., if two posits are equally near, take the one with binary encoding ending in 0. This ensures obtaining bitwise identical results across systems, which is not possible in the IEEE 754 since it defines up to five different rounding rules.

2.4.4 Fused operations and quire

In computer arithmetic, performing any operation the result if often rounded to fit in the original format. This may cause that computations involving two or more operations lose accuracy due to rounding of intermediate results. In order to avoid this problem, the rounding could be deferred until the last operation in computations that involve more than one operation. These are known as *fused operations*. The most common operation of this kind is the fused multiply-add (FMA), which computes the product of two numbers and adds the result to a third number in one step, with a single rounding. The FMA can speed up and improve the accuracy of many computations that involve the accumulation of products, such as dot products, matrix

multiplications or convolutions [11]. It was not until the revision of the IEEE 754 standard in 2008 [2], that the standard included the FMA operation in its requirements. According to the posit standard [12], fused expressions must be distinct from non-fused expressions in source code. For performing fused operations, the posit format introduces the concept of *quire*: a fixed-point format capable of storing exact sums and differences of products of posits. For each posit precision, there is also a quire format of precision depending on n and es parameters, as shown in Table 2.2. Thus, it is possible to consecutively add or subtract products of two posits without rounding or overflow. In contrast with previous accumulators, such as the Kulisch accumulator [13] the quire must be accessible to the programmer, so it might be possible to perform load/store and basic arithmetic (addition/subtraction) operations on it.

TABLE 2.2: Quire size for the different posit configurations

Posit$\langle n, es\rangle$	$\langle 8,0\rangle$	$\langle 16,1\rangle$	$\langle 32,2\rangle$	$\langle 64,3\rangle$	$\langle 128,4\rangle$
Quire size (bits)	64	256	512	2048	8192

2.4.5 Fast and approximate operations

The lack of fixed-length fields in the posit format hinders the decoding stage that precedes the core of each operation, in contrast with floating-point numbers. However, in addition to the properties already mentioned, there exist multiple operations that, under certain circumstances, can be done at bit level, without decoding the posit numbers [14]. Although some of such operations are approximations of complex functions, the results are quite similar to the original functions, and the reduction in time and resources is significant (just an ALU is enough for majority of cases). While it might be necessary to evaluate each application to decide if this approach provides a good trade-off, posits might be a suitable alternative for low-requirements devices running error-tolerant applications, such as embedded and IoT devices designed for AI and edge computing.

Some of the following operations are only valid for posit configurations with $es = 0$. In such case, Equation (2.5) can be written as $X = (-1)^s \times 2^k \times (1 + f)$.

Twice and half approximate operators

For any Posit$\langle n, 0\rangle$, the double of its value ($2X$) can be extremely well approximated without performing a common multiplication, which usually implies decoding the regime and fraction fields. Instead, for a posit bitstring p representing the real value X, it results that

$$2X \approx \begin{cases} p \ll 1 & \text{if } |X| \leq 0.5 \\ p \oplus (3 \ll (n-3)) & \text{if } 0.5 < |X| < 1 \\ ((signmask \wedge p) \vee (p \gg 1)) \oplus (1 \ll (n-2)) & \text{if } |X| \geq 1 \end{cases} \tag{2.6}$$

where $p \ll k$ represents a logical shift of bitstring p by k bits to the left (or to the right, if $\gg$, not preserving the sign bit in this case), $\oplus$ stands for bitwise xor, and *signmask* is a bit mask for the posit sign, i.e., `10...0`. As mentioned before, the result of this approximation is quite similar to the exact double value obtained by multiplication algorithm. In fact, Equation (2.6) returns the exact value for the first

two cases, where $|X| < 1$, and only fails when X is the maximum representable value (returning NaR) and on the third case by a posit away, when the rounding rules increment the final answer by 1. Thus, the approximated Equation (2.6) can be slightly modified, adding 1 in the third case when posit p has the two rightmost bits equal to 1, to provide a fast way to compute the exact double value of a posit. Although it might seem difficult to distinguish between the cases in Equation (2.6), for Posit$\langle n, 0\rangle$ configurations it just consists in comparing the first three bits, if they are all equal, then the first case should be applied, when just the first two match, it corresponds to the second case, otherwise third case corresponds.

Analogously, it is possible to approximate with almost no error the half of a given value ($X/2$) by bit-level operations:

$$X/2 \approx \begin{cases} (signmask \wedge p) \vee (p \gg 1) & \text{if } |X| \leq 1 \\ p \oplus (3 \ll (n-3)) & \text{if } 1 < |X| < 2 \\ (p \oplus (1 \ll (n-2))) \ll 1 & \text{if } |X| \geq 2 \end{cases} \tag{2.7}$$

In a similar manner as with the double operator approximation, Equation (2.7) only produces incorrect results in the first case (just by a posit away) and for the minimum representable values. Therefore, one can obtain the exact half value of almost any posit p from (2.7) by adding one to the result when $|X| \leq 1$ and the two rightmost bits of p are equal to 1.

Complement modulo 1

For any Posit$\langle n, 0\rangle$ between 0 and 1, that is $X \in [0, 1]$, its complementary value in that range $(1 - X)$ can be obtained by

$$1 - X = (1 \ll (n-2)) - p, \tag{2.8}$$

where p is the posit bitstring representing value X, so Equation (2.8) can be computed as for integer arithmetic.

Fast reciprocal operator

The multiplicative inverse of a posit value $(1/X)$ can be approximated by

$$1/X \approx (p \oplus \neg signmask) + 1, \tag{2.9}$$

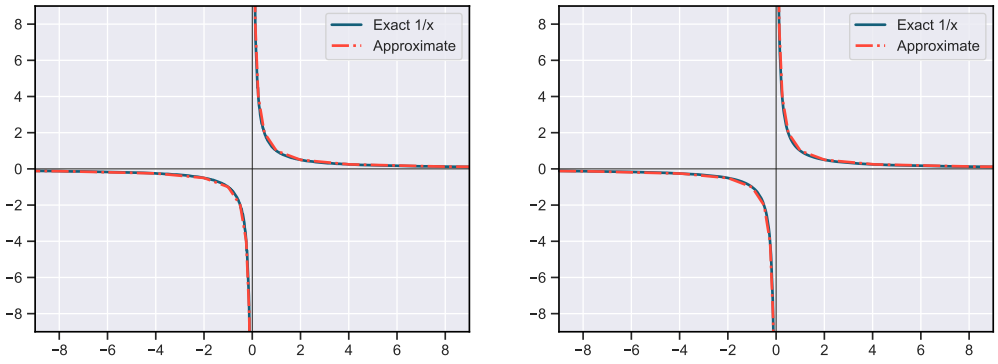
where $\neg$ stands for bit negation. The approximated result from Equation (2.9) is quite close to exact reciprocal value, as depicted in Figure 2.6.

Fast sigmoid activation function

The sigmoid function, $\sigma(x) = 1/(1 + \exp(-x)$, is one of the most commonly non-linear activation functions used in NNs. This function, as explained in [4], can be approximated with posits having $es = 0$ in a manner that avoids division and exponential operations, as Equation (2.10) shows.

$$\sigma(X) \approx (p \oplus signmask) \gg 2 \tag{2.10}$$

As can be seen in Figure 2.7, larger errors are far away from the critical region $x = 0$. The approximated sigmoid function can be used in combination with fast approximated operators of posit arithmetic seen before for other activation functions.



(A) Posit⟨8, 0⟩ (B) Posit⟨16, 1⟩

FIGURE 2.6: Fast reciprocal approximation function for different posit formats

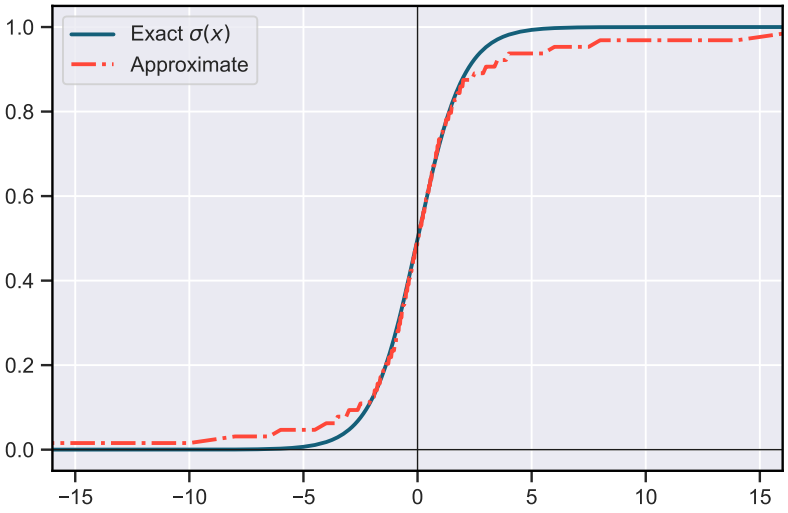


FIGURE 2.7: Comparison between exact and approximated versions of the sigmoid function using Posit⟨8, 0⟩

Fast extended linear unit

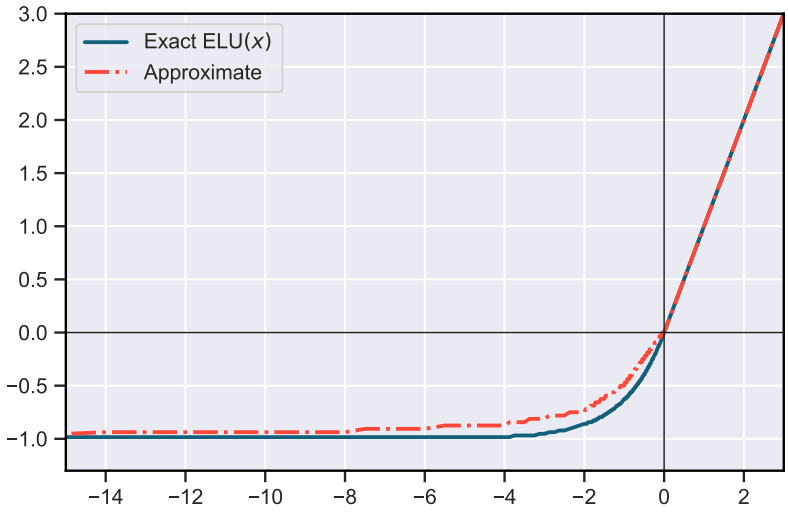
The extended linear unit (ELU) is another activation function that provides even better results than common rectified linear unit (ReLU) [15]. With $\alpha > 0$, the ELU function is defined by Equation (2.11).

$$\text{ELU}(x) = \begin{cases} x & \text{if } x >= 0 \\ \alpha(\exp(x) - 1) & \text{if } x < 0 \end{cases} \tag{2.11}$$

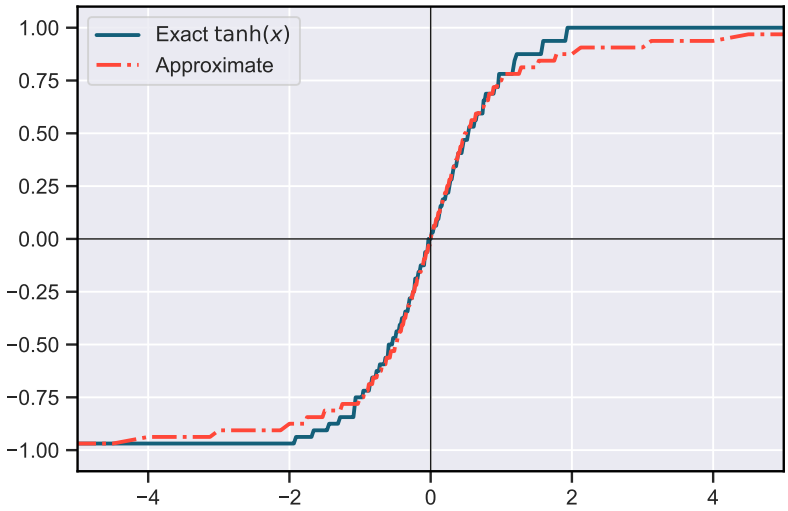
For the particular case of $\alpha = 1$, one can check that the negative part of the function can be rewritten as a linear combination of the sigmoid, as shown in Equation (2.12).

$$\exp(x) - 1 = -2\left(1 - \frac{1}{2\sigma(-x)}\right) \tag{2.12}$$

Therefore, the ELU function can approximated by using all the previous fast approximation functions. As can be seen in Figure 2.8a, the fast approximated values are really close to the real ones.



(A) ELU function



(B) Hyperbolic tangent

FIGURE 2.8: Comparison between exact and approximated versions of activation functions using Posit$\langle 8, 0\rangle$

Fast hyperbolic tangent

The hyperbolic tangent is another well-known activation function for NNs. Since the logistic sigmoid function is symmetric around the origin and returns a value in range $[0, 1]$, one can write $1 - \sigma(x) = \sigma(-x)$. Now, the expression of hyperbolic

tangent function can be rearranged in terms of the sigmoid function as follows:

$$\begin{aligned}\tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \\ &= 1 + \frac{-2e^{-x}}{e^x + e^{-x}} \\ &= 1 - \frac{2}{e^{2x} + 1} \\ &= 1 - 2\sigma(-2x) \\ &= 1 - 2(1 - \sigma(2x)) \\ &= 2\sigma(2x) - 1 \\ &= -(1 - 2\sigma(2x))\end{aligned} \tag{2.13}$$

In order to use the previous approximated functions, one must ensure that the necessary conditions are satisfied. One can see that, in general, the value obtained from $2\sigma(2x)$ is not in the unit interval. However, it is if $x \leq 0$, so for those values, it follows from Equation (2.13) that the hyperbolic tangent function can be approximated from a linear combination of the approximated sigmoid function, the twice and the one's complement operators. Finally, thanks to the the antisymmetry of the tanh function, this reasoning can be extended to positive values, as Figure 2.8b shows.

2.5 Drawbacks of the posit format

Although multiple advantages of the posit format have been shown so far, there are a number of disadvantages with respect to other traditional decimal arithmetic formats such as floating point and fixed point.

The main problem associated with this format is the run-time variable-length regime field. Since it does not have a fixed length, as the exponent does in the floating-point format, it is not possible to decode the fields of a posit number in parallel, as is the case with the IEEE 754 format. This implies certain overhead from the point of view of hardware design. However, the proponents of this format claim that the fact that posits with fewer bits present similar precision as floating-point numbers compensates for this drawback.

Besides, there are certain domain-specific problems, such as particle physics simulation, where posits provide worst results than floats, mainly due to the error from computations based on physical constants [16]. Posits have higher accuracy than floats on an interval around 0, but lower outside it, and many common values used in physics, such as Plank and Boltzmann constants, Avogadro number or the speed of light are outside that interval. A possible solution to this could be to scale the entire calculation by the physical constant, or to do calculations with logarithms.

Finally, the cost of hardware development for the posit format, together with the lack of software and compilation tools, slow down the research of this numeric format, and makes it difficult to assess its progress.

3 Evaluation of Posit Arithmetic in Deep Neural Networks[1]

In Section 2.4, many interesting properties of posits that can be exploited in the area of Deep Neural Networks (DNNs) were presented. In this chapter, this idea is carried out to check the benefits of the posit format. After a brief introduction to DNNs, experiments on training and low precision inference are presented. The performance of different posit formats is compared with the baseline floats on multiple networks and datasets. For this purpose, a deep learning framework based on the PNS has been developed.

Section 3.1 presents the most relevant concepts of DNNs, including convolutional networks, which are one of the most used techniques in modern computer vision. In Section 3.2 the state of the art in posit arithmetic and its applications in NNs are revised. The developed framework to run the experiments using posits is explained in detail in Section 3.3. Finally, Section 3.4 details the multiple experiments carried out, as well as the results and conclusions obtained from them.

3.1 Background on deep neural networks

Within the broad field of AI is a large subfield called machine learning, which studies how to give computers the ability to learn without being explicitly programmed. That means this kind of programs, once created and after a process called training, will be able to learn how to do some intelligent activities outside the notion of programming [18].

Within the machine learning field, deep learning is a set of brain-inspired methods and algorithms that tries to learn representations of data with multiple levels of abstraction through models composed of multiple layers of processing called deep neural networks or DNNs [19]. These methods have dramatically improved the state of the art in speech recognition, visual object recognition, object detection, and many other domains such as drug discovery and genomics. Typically, deep learning tasks are computationally expensive, specially during the training phase, which requires substantial energy consumption. In fact, GPUs have been the key component for much DNN processing in the last decade, and there is an increasingly interest in providing more specialized acceleration of the DNN computation [6], [7], [18]. This, together with the new trends of the IoT and edge computing, gives rise to a paradigm shift in which small data processors are incorporated into the sensors, so that the "intelligent" devices themselves are the ones responsible for generating responses (almost immediately) to the events produced. It is for this reason that large

[1]The content of this chapter is originally published in Digital Signal Processing: A Review Journal [17]. The manuscript has been reformatted for inclusion in this MSc thesis.

companies in the hardware industry such as Nvidia or Intel are investing in powerful embedded High-Performance Computing (eHPC) platforms that allow acceleration of neural networks in real-time IoT devices such as drones or autonomous vehicles [20].

3.1.1 Neural networks

Artificial neural networks, or NNs, take their inspiration from how the brain works. It is generally believed that the main computational element of the brain is the *neuron*. Each neuron accepts signals as inputs, performs a computation on those signals, and generates an output signal as response. These signals are referred to as *activations*. The input activations are scaled by a factor, which is known as *weight*, and the way the brain is believed to learn is through changes to these weights. This computation is emulated in artificial neurons, the basic building-blocks for artificial neural networks. Figure 3.1 illustrates how these neurons work, they compute the output activation y as a weighted sum of the input activations x_i and the weights w_i plus a bias b and passing the result through a nonlinear function $f(\cdot)$, referred to as *activation function*.

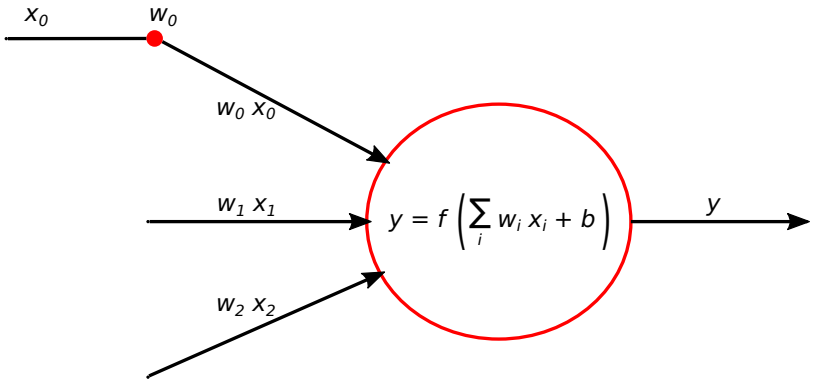


FIGURE 3.1: Operation of an artificial neuron

The simplest NNs are made up of several layers, which in turn consist of different number of artificial neurons, as shown in Figure 3.2. These are also known as *feedforward* networks. The neurons in the input layer receive some values and propagate them to the neurons in the middle layer of the network, which is also frequently called a "hidden layer". In a similar manner, the weighted sums from the hidden layer are propagated to the output layer, which presents the final outputs of the networks, for example, as probabilities for each of several categories.

As can be seen, this kind of networks propagate the output of each layer to the input of the following. This is the reason that each neuron does not just output a weighted sum, but applies a nonlinear activation function to it, since the computation obtained from a sequence of neurons would then be a simple linear algebra operation. Currently, the most popular nonlinear function is the Rectified Linear Unit (ReLU), which is simply the half-wave rectifier $f(z) = \max(z, 0)$, although there are many others, such as the sigmoid or the hyperbolic tangent, discussed in Section 2.4.

3.1.2 Deep neural networks

Within the domain of neural networks, there is an area called *deep learning*, in which the networks have more than three layers, i.e., more than one hidden layer. These kind of networks are frequently called deep neural networks or DNNs. Nowadays,

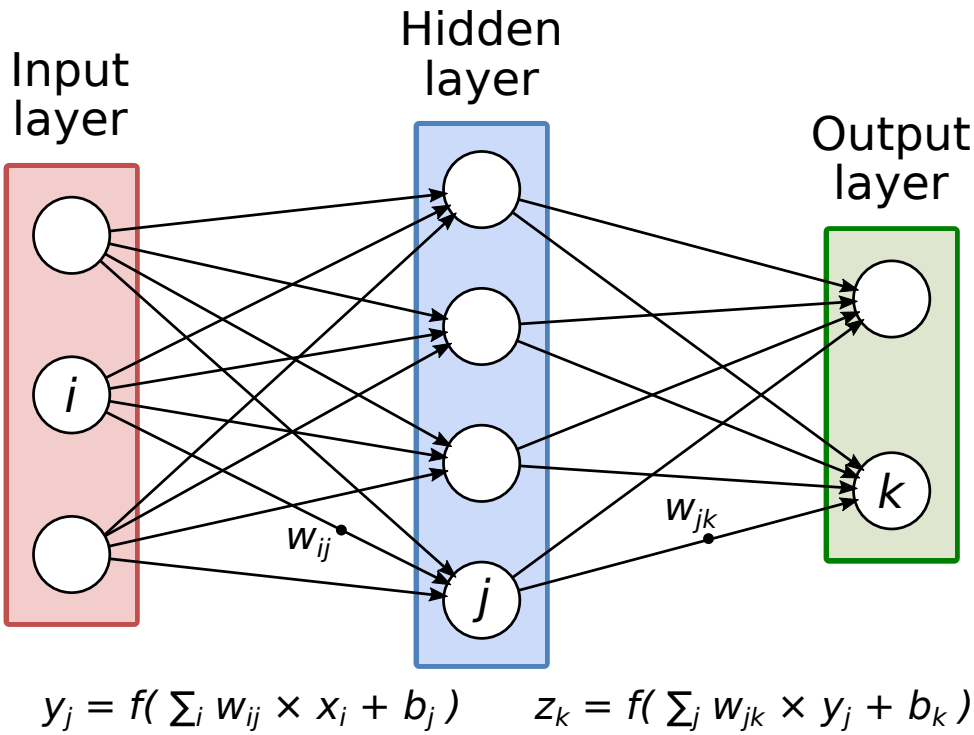


FIGURE 3.2: Simple neural network diagram

the typical number of network layers used in deep learning ranges from five to more than a thousand [21].

DNNs can have a huge number of neurons, so calculating the weights by hand would be impractical. For this reason, in the mid-1980s, after seeing those multilayer architectures could be trained by descending the gradient, a process based on the chain rule for derivatives called *backpropagation* was proposed [22]. This algorithm allows recalculating each one of the weights that are used in each neuron in an iterative way, which is known as the *training* phase of the neural network. Once trained, the network can compute an output value using the weights determined during the training process, without recalculating them. This is referred as the *inference* or prediction phase. The backpropagation computation is, in fact, very similar in form to the computation used for inference, but the former has higher requirements, since it is necessary to preserve intermediate outputs of the network and generally with higher precision [23], [24].

As can be seen, the operations of every single neuron mainly consist of weighted sums, and therefore, when thousands of them are stacked in multiple layers to form DNNs, it is clear that DNN computation is mostly matrix multiplications. For this reason, it is quite common to use parallelizaton techniques such as SIMD or SIMT and accelerators like GPUs that optimize for matrix multiplications.

Depending on the application, DNNs come in a wide variety of shapes and sizes. The aforementioned feedforward networks are probably the simplest kind of networks, where all of the computation is performed as a sequence of operations on the outputs of a previous layer. An alternative to this are recurrent neural networks, in which some intermediate values are stored internally and used as inputs to other operations in conjunction with the processing of a later input. The layers that compose the DNNs can also be of different kinds. In an *fully connected* (FC) layer, all output activations are composed of a weighted sum of all input activations, as hidden layer in Figure 3.2. If not all the connections between the activations are present, the layer

is called *sparsely connected*. In addition to FC layers, various kinds of layers can be found in a DNN such as the aforementioned nonlinearity, convolutional and pooling (which will be explained in next section) and normalization.

3.1.3 Convolutional neural networks

Nowadays, one of the most common techniques in deep learning is the use of Convolutional Neural Networks (CNNs) [25], [26]. These networks have shown exceptional inference performance when trained with large amounts of data, and, in fact, problems where humans used to perform better than machines, such as image classification, can now be solved with such methods, outperforming human accuracy. CNNs are designed to process data given in the form of multidimensional vectors (or tensors), so they are especially useful in tasks such as computer vision, classification and segmentation of videos and images or audio signal and speech recognition.

The typical architecture of a CNN is structured as a series of stages. The first stages, called feature extraction, are made up of two types of layers: *convolutional* layers and *pooling* layers. These stages are stacked, followed by FC layers, which fulfill the classification function.

The convolutional layer is the central building block of a CNN. The units in a convolutional layer are organized into high-dimensional convolutions, each of which is made up of a small set of weights called filters or *kernels* that span across the total depth of the input activations or *feature maps*, each of which is called a *channel*. In the forward (or inference) step of a convolutional layer, these kernels perform the convolution operation indicated by Equation (3.1)

$$X_j^{L+1} = g\left(b_j + \sum_i K_{ij} \circ X_i^L\right), \tag{3.1}$$

where X_j^{L+1} is the j-th output feature map of the current layer L, which is computed as the sum of products of the kernels K_{ij} and the input feature map X_i^L obtained from previous layer, then adding the bias b_j, and finally applying the non-linear activation function g. Mathematically, the operation performed by a filter is a discrete convolution, hence the name. Figure 3.3 illustrates this operation for the case where the input feature map is an RGB color image (thus, it will contain three channels that correspond to the red, green, and blue components).

While the function of the convolutional layers is to extract the local characteristics of the previous layer, the pooling layers are used to merge semantically similar characteristics into one, as a non-linear compression. The compression or pooling operation is specified (rather than learned) and applied as a filter to feature maps, usually with a size smaller than those maps. The two most common functions in pooling operations are:

- **Average Pooling**: Calculate the average value for each patch on the feature map.
- **Maximum Pooling (or Max Pooling)**: Calculate the maximum value for each patch of the feature map.

Pooling layers reduce the dimension of feature maps and are often stacked just after convolution layers.

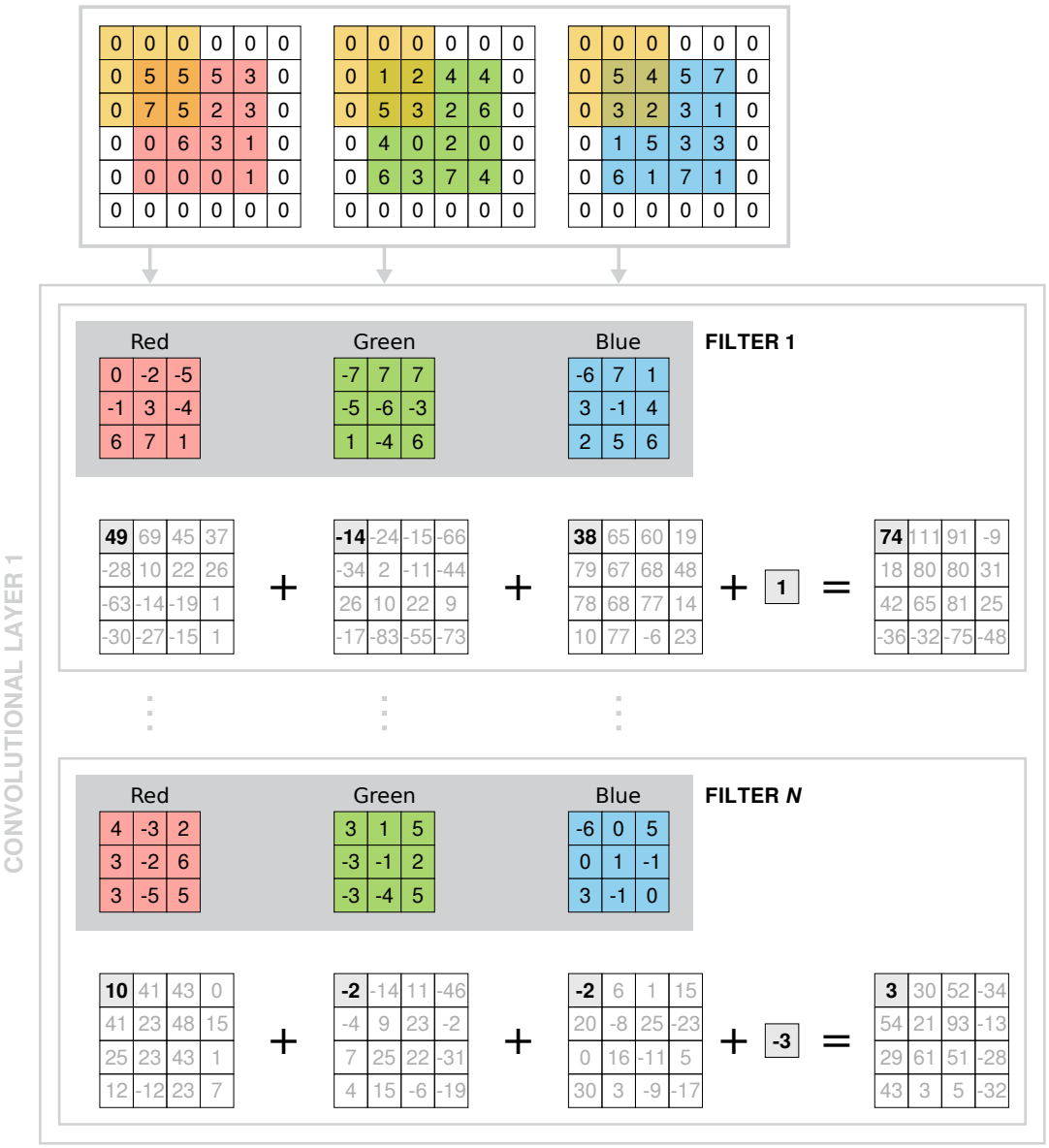


FIGURE 3.3: Convolution operation in a CNN

3.2 Related work

Since introduction of posits in 2017, multiple works have explored the benefits of this novel format against the standard floating point, and most of them focusing on NNs. The main problematic with posits is that there is currently no available hardware support for this format, and majority of works perform arithmetic operations via software emulation. In [27], J. Johnson designed an arithmetic unit for combining posit addition together with logarithmic multiplication [28], [29] for performing CNN inferences, but just in simulation. Other studies tackling the inference stage of CNNs are performed in [30]–[33]. Authors in [30], [31] employed a posit DNN accelerator to represent weights and activations combined with an FPGA-based softcore for 8-bit posit FMA operations. They demonstrated that 8-bit posits outperform both 8-bit fixed-point and floating-point numbers on low-dimensional datasets. This work was later extended in [32], [33]. In all these works, the training stage is performed in floating point, while the inference stage is performed in low-precision

posit format.

In order to avoid this conversion and show the whole potential of posits, it would be desirable to perform training using posit format. So far today, only few recent works [34]–[37] perform this phase entirely with posits. Nonetheless, [37] relies on 32-bit floats for initial warm-up training, while the others just deal with feedforward neural networks, which are simpler than CNNs.

This work presents Deep PeNSieve, a framework to entirely train DNNs using posits during the whole process, and besides, to perform low-precision inference with 8-bit posits. To the best of my knowledge, this is the first work proposing the training of CNNs entirely using the PNS. Furthermore, the CIFAR-10 dataset will be evaluated as well, which is more complex than the datasets studied in the aforementioned works.

3.3 The framework: Deep PeNSieve

In this work, a novel posit-based framework for DNNs called Deep PeNSieve is proposed. This framework allows to entirely perform both training and inference of deep neural networks employing the PNS. Trained models are saved preserving the format in order to perform inference with the same or even lower precision. As already mentioned, posits possess interesting properties that may be exploited particularly in the DNNs domain.

Deep PeNSieve is build on top of the well known machine learning framework TensorFlow [38], which implements all the functionality necessary to develop NN models, such as different layers, forward and backward algorithms, optimizers, etc. The posit number format is extended to this framework via software emulation. For the quire support of the framework (only for the inference stage), Deep PeNSieve relies on the SoftPosit library, and implements the rest of algorithms necessary for inference. Therefore, the whole framework is written in Python programming language. Let us now describe in detail how training and inference are performed.

3.3.1 Training interface

The training flow of the proposed framework is depicted in Figure 3.4. Red boxes represent input data, including hyperparameters, while the blue boxes correspond with the functionalities performed by the TensorFlow framework. In the first place, it must be noted that all the inputs of networks must be in posit format and, therefore, the outputs will be in such format too. When creating a new DNN (which may include convolutional and fully-connected layers, but not only), all the hyperparameters are initialized employing the posit format. To train the model, in this case using the Adam method [39] (but any other optimizer like SGD, Momentum or Nesterov could be selected), the optimizer parameters must be also converted to the posit format. At the time the network is generated using only posits with the selected $\langle n, es \rangle$ configuration, it is trained as usual, with the single difference that all computations are performed with posit numbers, which allows us to evaluate the true performance of such format in this task. In this manner, the trained parameters and models are saved to perform the inference stage with the same or lower precision.

Finally, it is interesting to point that Deep PeNSieve performs posit number computations via software emulation, which provokes an overhead regarding the baseline single-precision floating-point (float32) training cases. Therefore, comparing

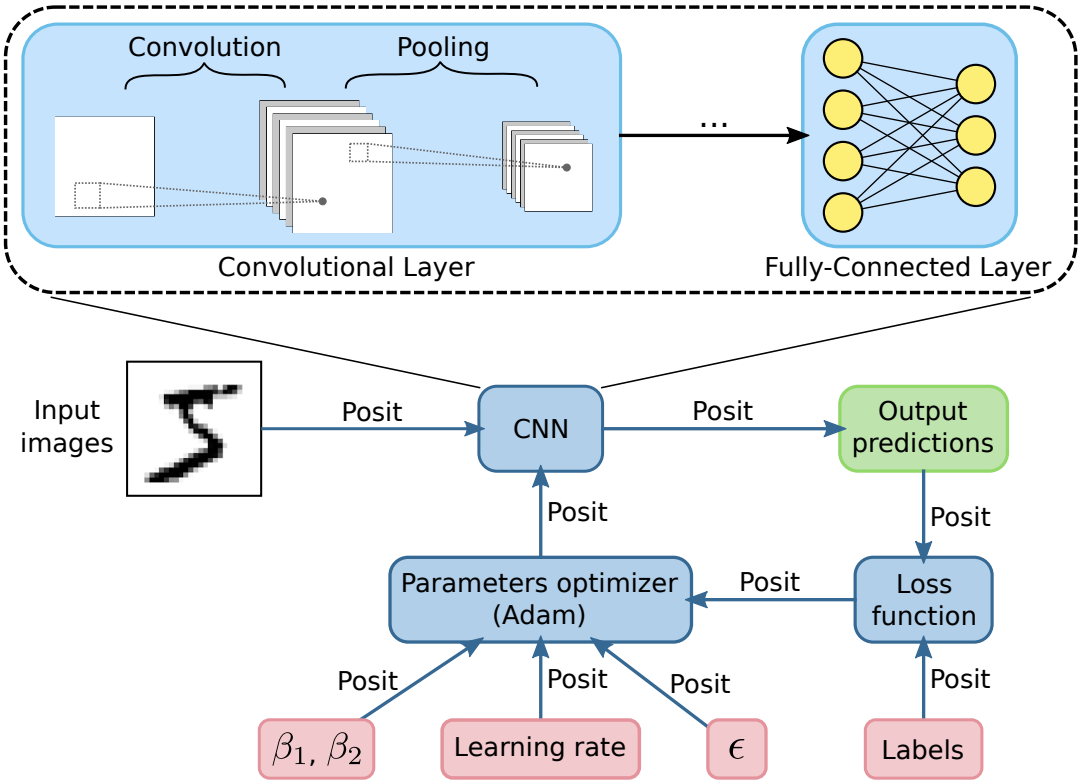


FIGURE 3.4: Illustration of the training flow of the proposed framework

training times between different numeric formats is beyond the scope of this work, as the lack of dedicated hardware is a further handicap for the PNS.

3.3.2 Low precision posits for inference

Post-training quantization is a widely used conversion technique consisting of storing tensors at a lower bitwidth and which can reduce the model size while also improving latency, with little degradation in model accuracy. Typical lower-precision numerical formats used in quantization are 8-bit integers (INT8) [40], half-precision floating-point format [23] and brain floating-point format (bfloat16 or BF16) [8]. Nonetheless, the majority of literature on neural network quantization involves either training from scratch [23], [41] or fine-tuning the pre-trained models [40], [42].

Deep PeNSieve allows post-training Posit$\langle 8, 0\rangle$ quantization. The stored model can be converted to a low precision one by keeping the model architecture and converting the original parameters to Posit$\langle 8, 0\rangle$ format so that the operations of the model are virtually not changed, only the data format. The proposed solution consists in performing a linear quantization of model parameters to lower posit precision format as (3.2) shows.

$$\tilde{x} = round(clip(x, minpos, maxpos)) \,, \tag{3.2}$$

where x is the original value, $round()$ performs rounding according to selected precision, $minpos = useed^{2-n}$ is the smallest nonzero value expressible as a posit in such precision, $maxpos = useed^{n-2}$ is, analogously, the largest real value expressible as a

posit and function $clip$ is defined by (3.3).

$$clip(x, minpos, maxpos) = \begin{cases} sign(x) \times maxpos & \text{if } |x| > maxpos \\ sign(x) \times minpos & \text{if } |x| < minpos \\ x & \text{otherwise} \end{cases} \tag{3.3}$$

It is worthy to note that posit arithmetic does not underflow or overflow, and clipping posits should be implicitly made by either hardware or software. Eventually, note that it is not needed to quantize activations since only precision is reduced, while arithmetic is maintained. This is an advantage in contrast with classical quantization processes: when quantizing floating-point models to 8-bit precision, the dynamic range of unbounded activations, such as ReLU, needs to be calculated using calibration data [40].

3.3.3 Fused dot product approach

The current trend when quantizing a trained neural network is using INT8 as low precision format. However, arithmetic operations in quantized neural networks using 8-bit integers requires INT16 or INT32 in practice, since INT8 can barely hold the result of certain operations such as multiplication and addition. In particular, General Matrix Multiply (GEMM), which is the core operation of NNs, involves lots of multiplications and additions and many deep learning frameworks use 32-bit accumulators for intermediate results [40]. Furthermore, novel techniques for training DNNs with low precision arithmetic [24], [43] make use of higher (16 or 32) bitwidth accumulators to maintain model accuracy across GEMM functions during forward and backward passes. This suggests that quantized 8-bit posit models would require an additional structure to preserve accuracy. As it has been mentioned, the quire register is an accumulator included in the draft posit standard [12] specially designed for fused operations. Therefore, it would be interesting to use the quire for the fused dot product operations of GEMM for both convolutional and fully connected layers of 8-bit posit neural networks.

For that purpose, the internal structure of such layers must be modified, so that the GEMM function is performed with the fused dot product. Although the TensorFlow framework provides useful tools for designing and training NNs, posit fused operations are unsupported at the moment of writing. The SoftPosit[2] reference library, however, includes posit fused operations employing quire accumulator. It is necessary to re-implement the same network architecture with this library accordingly for post-training quantization using Posit$\langle 8, 0\rangle$ with quire. In particular, for an 8-bit length posit, the draft standard specifies that the size of the quire is 32 bits. Figure 3.5 shows how, for a given layer L, the GEMM functions are implemented using the quire during the forward stage and, for the sake of completeness, Algorithm 1 describes this kernel modification using a single quire accumulator. Note how the intermediate results are stored in a quire, so just one rounding is performed at the end of each computation.

The GEMM function is inherited in fully connected layers, but not so for the convolutions. The convolutional layers are the most computationally intense parts of CNNs. Currently, a common approach to implement convolutional layers is to expand the image into a column matrix (im2col method) and the convolution kernels

[2]Source code accessed on January 10, 2021 from gitlab.com/cerlane/SoftPosit.

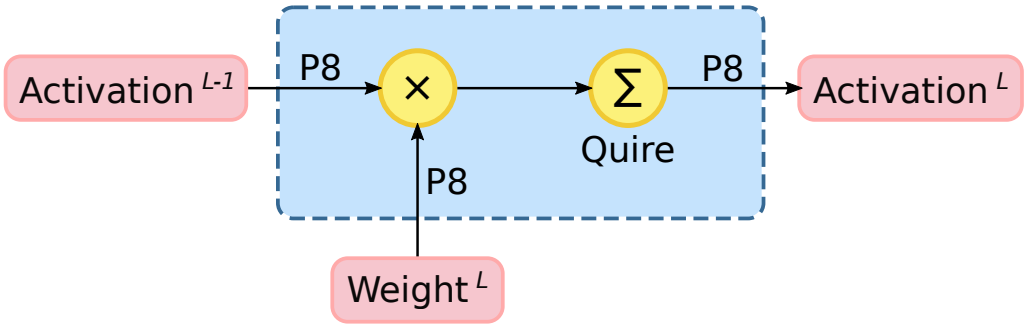


FIGURE 3.5: Low precision data flow for GEMM function in forward propagation

Algorithm 1 GEMM function with quire

Require: $X \in \mathcal{M}_{n\times l}$, $W \in \mathcal{M}_{l\times m}$, in posit format
Ensure: $Y = X \times W \in \mathcal{M}_{n\times m}$
for all $i \in [0, n]$ **do**
  for all $j \in [0, m]$ **do**
    $q \leftarrow quire(0)$ // Quire initialization
    for all $k \in [0, l]$ **do**
      $q \leftarrow x_{i,k} \cdot w_{k,j} + q$ // The result is accumulated, but not rounded
    end for
    $y_{i,j} \leftarrow posit(q)$ // The operation is rounded here
  end for
end for

into a row matrix. This way, convolutions can be performed as matrix multiplications. This work exploits this approach to benefit from the quire-based implementation of GEMM in convolutional layers.

3.4 Experimental results for DNN training and inference

To check the performance of the different numeric formats, training and inference results for several CNN architectures, datasets, and precision are compared. As in most Deep Learning (DL) frameworks, training is done under IEEE 754 single-precision floating-point format, while half-precision floats or 8-bit integers are used for low precision inference. With regard to the PNS, J. Gustafson recommends the use of Posit$\langle 32, 2\rangle$ and Posit$\langle 16, 1\rangle$ configurations, arguing that such 16-bit posits could be used to replace 32-bit floats [4]. Therefore, training CNNs with the aforementioned posit configurations will be compared with the baseline Float 32. Note that many DL frameworks use by default 32-bit floats for training (some support the so-called mixed-precision training [23]), while half-precision or 8-bit integers can be used just for inference (this is the case, for example, of PyTorch or TensorFlow, which is used in this work). Deep PeNSieve is employed to handle the training process with posits and TensorFlow for the rest of the formats.

With regard to Posit$\langle 8, 0\rangle$ configuration, no convergence was obtained in the training experiments. Similar results were obtained in [34]. Without further modifications on the training flow, such low precision is not suitable for this task. Nonetheless, accuracy results when performing post-training quantization from Posit$\langle 32, 2\rangle$

to Posit$\langle 8, 0\rangle$ (with and without the use of quire and fused operations) are compared to conventional Float 32-based quantization techniques.

Experiments were run on a computer with an Intel® Core™ i7-9700K processor (3.60 GHz) with 32 GB of RAM.

3.4.1 Benchmarks

To compare the benefit of posits for DL tasks, multiple datasets and CNNs architectures for image classification have been employed.

The datasets used for the training and testing of the different DNNs are the following:

- **MNIST**: Set of 28×28 pixel grayscale images with handwritten digits [25]. The dataset, with 10 categories and 7000 images per category, is divided into 60,000 training images and 10,000 test images.
- **Fashion-MNIST**: Set of 28×28 pixel grayscale images with photographs of fashion garments [44]. The dataset, with 10 categories and 7000 images per category, is divided into 60,000 training images and 10,000 test images. This set, with the same characteristics as MNIST, is considered more difficult than the previous one.
- **SVHN**: Set of 32×32 pixel color images from Google Street View photographs with house numbers [45]. With a total of 73257 digit images for training and 26032 for test, this dataset, although similar to MNIST, is much more complex than this one since it deals with real photographs and contains color images (RGB, 3 channels per image), which increases notably the number of operations to be carried out.
- **CIFAR-10**: Set of 32×32 pixel color images of various objects grouped into 10 different categories, with 6000 images per category [46]. The dataset is divided into 50,000 training images and 10,000 test images. This set is notably more complex than the previous ones due to the variability between objects of the same class.

On the other hand, the CNN architectures to use are the following:

- **LeNet-5**: Classic convolutional network designed by Lecun et al. [25] to classify the MNIST dataset. The original structure of this network, depicted in Figure 3.6, consists of two sets of convolutional and average pooling layers, followed by a flattening convolutional layer, then two fully-connected layers and finally a softmax classifier, which makes a total of $\sim 6.2 \times 10^4$ parameters. In this case, the average pooling functions will be replaced by max pooling, and the ReLU function will be used as activation instead of the hyperbolic tangent as in the original article. With this architecture, the training on the MNIST and Fashion-MNIST datasets will be carried out.
- **CifarNet**: Convolutional network originally designed to solve the CIFAR-10 classification problem [46]. In this case, a variant of this architecture will be used, changing the number of feature maps and the size of the pooling filters as shown in Figure 3.7. This architecture will be used with the SVHN and CIFAR-10 datasets, and contains $\sim 1.7 \times 10^6$ parameters.

Table 3.1 summarizes the different datasets, network architectures and training configurations used for the experiments in this work.

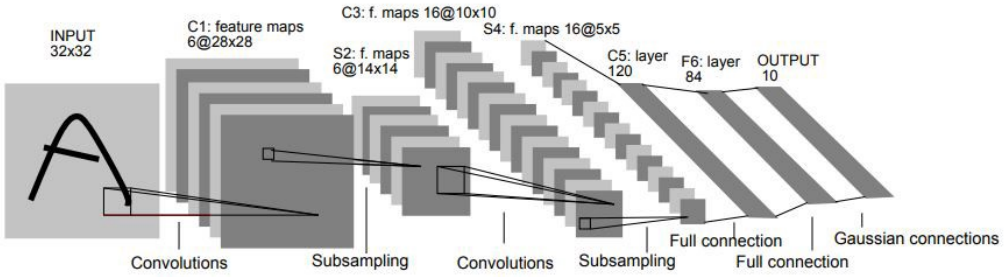


FIGURE 3.6: Original LeNet-5 architecture

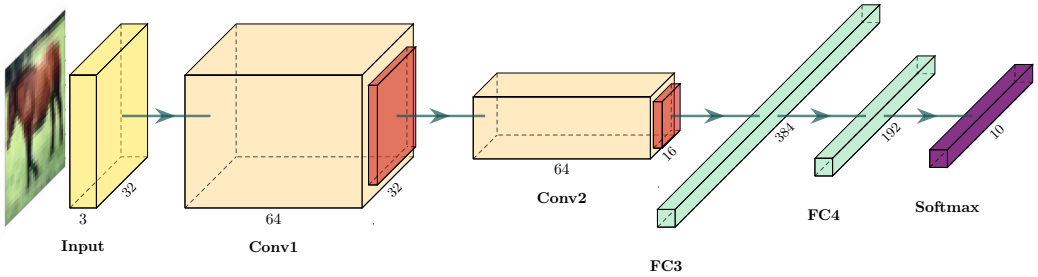


FIGURE 3.7: CifarNet architecture

TABLE 3.1: DNNs setup

Dataset	Architecture	Optimizer	Batch size	Epochs
MNIST	LeNet-5	Adam	128	30
Fashion-MNIST	LeNet-5	Adam	128	30
SVHN	CifarNet	Adam	128	30
CIFAR-10	CifarNet	Adam	128	30

3.4.2 DNN training results

The aforementioned architectures are implemented and trained with three different formats: float32 (default precision for training), Posit$\langle 32, 2\rangle$ and Posit$\langle 16, 1\rangle$. No regularization techniques, such as normalization and dropout, are used in the training process. Input data are normalized into the range $[-1, 1]$. Table 3.2 shows comparisons among the inference results of standard float32 and posits on different trained models. As can be seen, models trained with PNS present similar results than the baseline float32. Moreover, networks employing Posit$\langle 16, 1\rangle$ show higher accuracy than 32-bit formats, especially for the complex CIFAR-10 dataset, where Top-1 is more than 4% higher than that obtained with float32. While this is a very significant improvement, it does not mean yet that posits behave better than floats. For the sake of completeness, regularization techniques, such as batch normalization or dropout, should be ported to the PNS as well, although this escapes the scope of this work.

Besides the accuracy, it is important to note the model size reduction of low precision formats: while 32-bit models require 724 KB and 21 MB when stored on disk for LeNet-5 and CifarNet, respectively, models on Posit$\langle 16, 1\rangle$ format require just 362 KB and 10.5 MB, respectively. Thus, with the corresponding posit hardware support the use of 16-bits posits would reduce the memory usage of the NNs, enabling

training larger models or with larger mini-batches [23].

TABLE 3.2: Accuracy results after the training stage

	MNIST		Fashion-MNIST		SVHN		CIFAR-10	
Format	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
Float32	99.17%	100%	89.34%	99.78%	89.32%	98.35%	68.06%	95.15%
Posit⟨32, 2⟩	99.09%	99.98%	89.90%	99.84%	89.51%	98.36%	69.32%	96.59%
Posit⟨16, 1⟩	99.18%	100%	90.17%	99.81%	90.90%	98.72%	72.51%	97.40%

For a better understanding of the training process, Figures 3.8, 3.9, 3.10 and 3.11 illustrate it along the different epochs for MNIST, Fashion-MNIST, SVHN and CIFAR-10 datasets, respectively. The CNNs implemented on the PNS converge in a similar manner as the floating-point ones. Nonetheless, networks on Posit⟨16, 1⟩ present lower error throughout training than other formats, which leads to higher accuracy results as mentioned before.

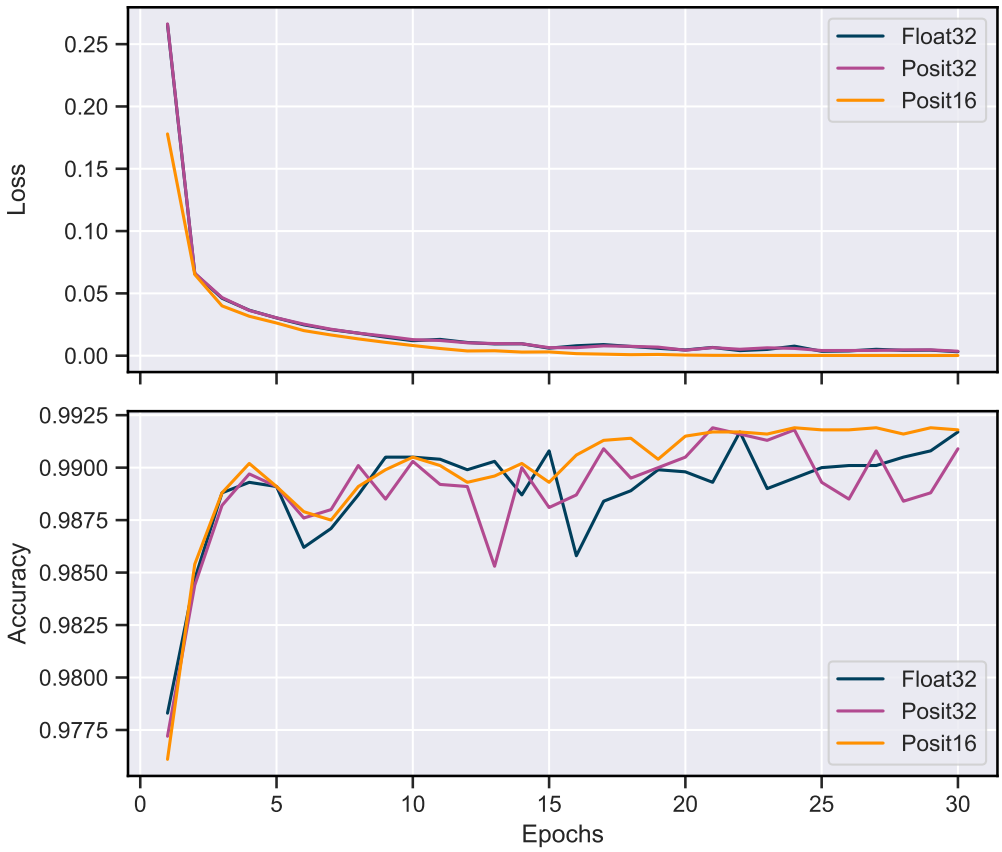


FIGURE 3.8: Learning process along LeNet-5 training on MNIST

Finally, as mentioned before, there is no dedicated hardware support for the posit format, and all the posit-based computations must be performed via software emulation. This has a great impact on the execution of the programs, as Table 3.3 illustrates. As can be seen, at this moment, posit arithmetic is not competitive against floating point in terms of computing times and speedup.

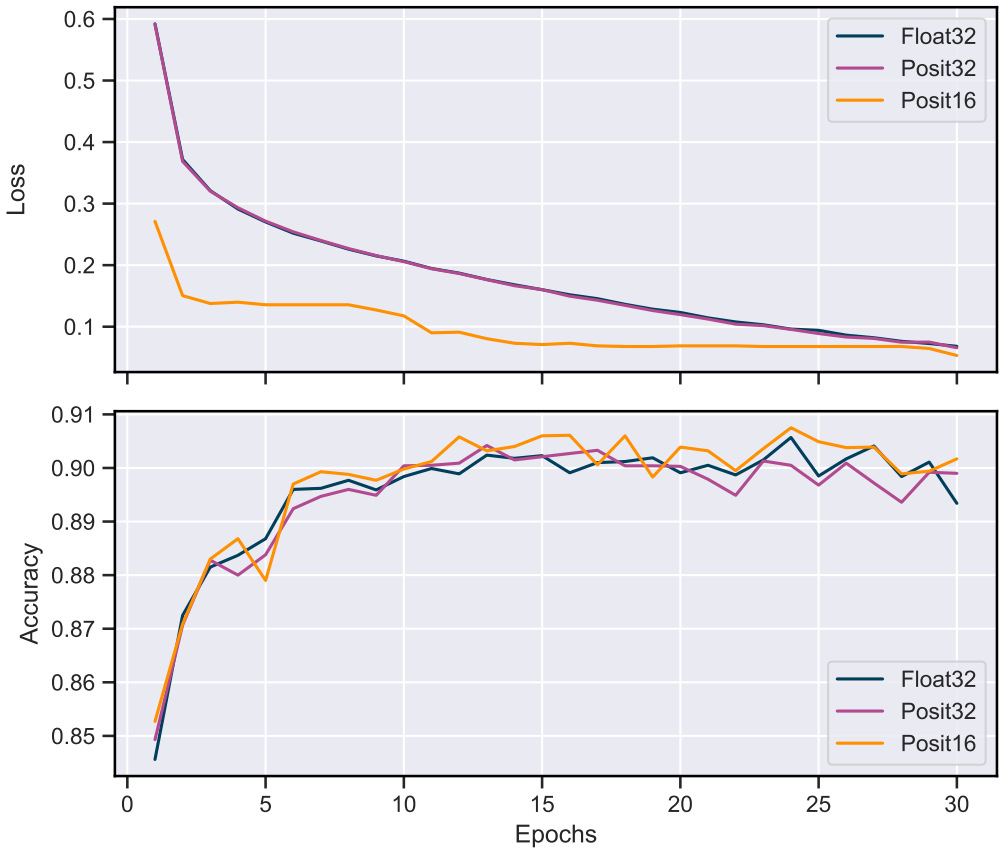


FIGURE 3.9: Learning process along LeNet-5 training on Fashion MNIST

TABLE 3.3: Training time for each of the models and numeric formats

	Time per epoch (HH:MM:SS)			
Format	MNIST	Fashion-MNIST	SVHN	CIFAR-10
Floating-Point	00 : 00 : 09	00 : 00 : 09	00 : 01 : 58	00 : 01 : 17
Posit	00 : 05 : 35	00 : 05 : 35	06 : 13 : 28	03 : 47 : 26

3.4.3 DNN post-training quantization results

The previously trained models are kept unchanged in order to perform low precision inference. Then, common NN quantization methods for float32 and Posit$\langle 32, 2\rangle$ models are compared. For the floating-point case, float16 quantization and integer quantization techniques are employed to obtain models that entirely work in float16 and INT8 formats, respectively. With regard to posits, the 32-bit models have been quantized to Posit$\langle 8, 0\rangle$, as described in Section 3.3.2. Moreover, this work compares inference results between naive posit quantization and employing the 8-bit posit fused dot product approach, which additionally requires changing convolutional and fully connected layers of the models as in Section 3.3.3.

Table 3.4 shows the inference results for the different post-training quantization techniques. Note that quantized 16-bit and 8-bit models are, respectively, 1/2 and 1/4 the size of the corresponding original 32-bit models. Results confirm the potential of conventional quantization techniques. Nevertheless, it is well-known that for

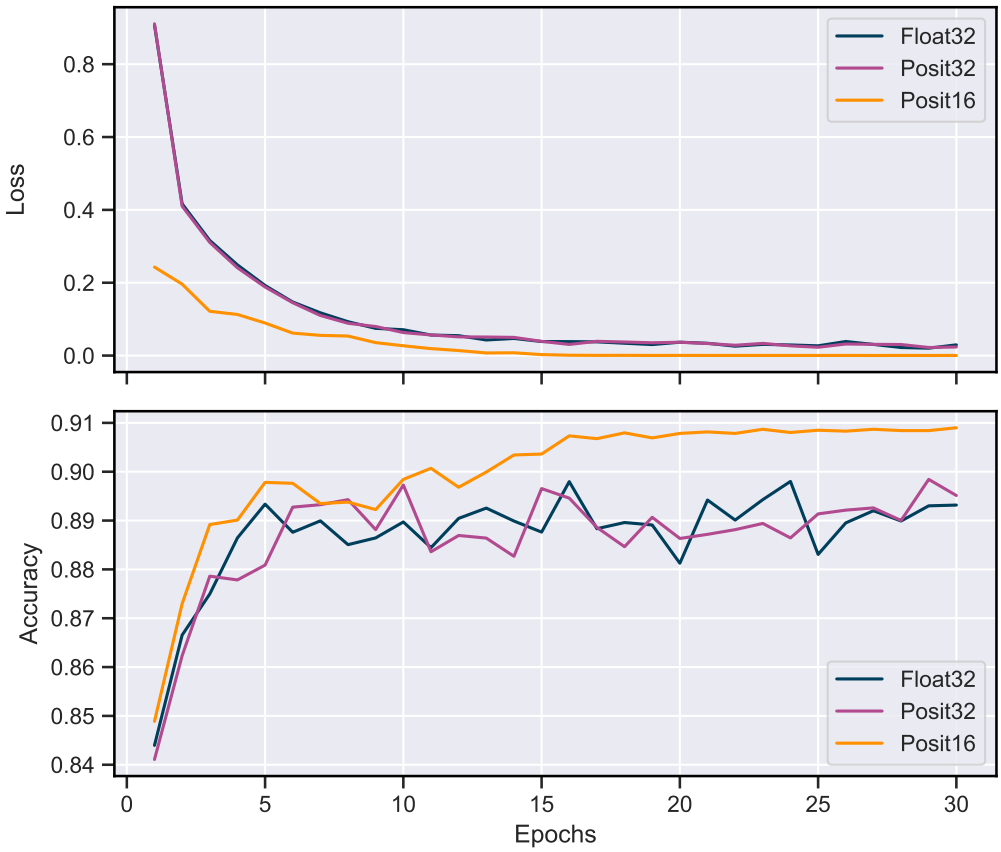


FIGURE 3.10: Learning process along CifarNet training on SVHN

larger networks such as MobileNet, integer quantization provides higher accuracy degradation [40]. On the other hand, there is a notable difference in the posits case between using quire and fused operations or not. Networks using fused dot product with quire get much higher accuracy (25% higher top-1 for CIFAR-10) as not using, and this difference becomes more noticeable with more complex networks. In the worst case, on CIFAR-10 dataset, accuracy degradation of Posit$\langle 8,0\rangle_{\text{quire}}$ is just 0.44% with respect the original 32-bits model, which is more than acceptable.

TABLE 3.4: Post-training quantization accuracy results for the inference stage

	MNIST		Fashion-MNIST		SVHN		CIFAR-10	
Format	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
Float16	99.17%	100%	89.34%	99.78%	89.32%	98.35%	68.05%	96.15%
INT8	99.16%	100%	89.51%	99.79%	89.33%	98.38%	68.15%	96.14%
Posit$\langle 8,0\rangle$	98.77%	99.99%	88.52%	99.82%	81.31%	97.07%	43.89%	86.49%
Posit$\langle 8,0\rangle_{\text{quire}}$	99.07%	99.99%	89.92%	99.81%	89.13%	98.39%	68.88%	96.47%

The proposed low precision posit approach provides similar results as the widely used post-training quantization techniques provided by the TensorFlow framework. Higher accuracy of PNS on Fashion-MNIST and CIFAR-10 datasets might be the result of the fact that initial models on Posit$\langle 32,2\rangle$ format present higher results than corresponding float32 models as well.

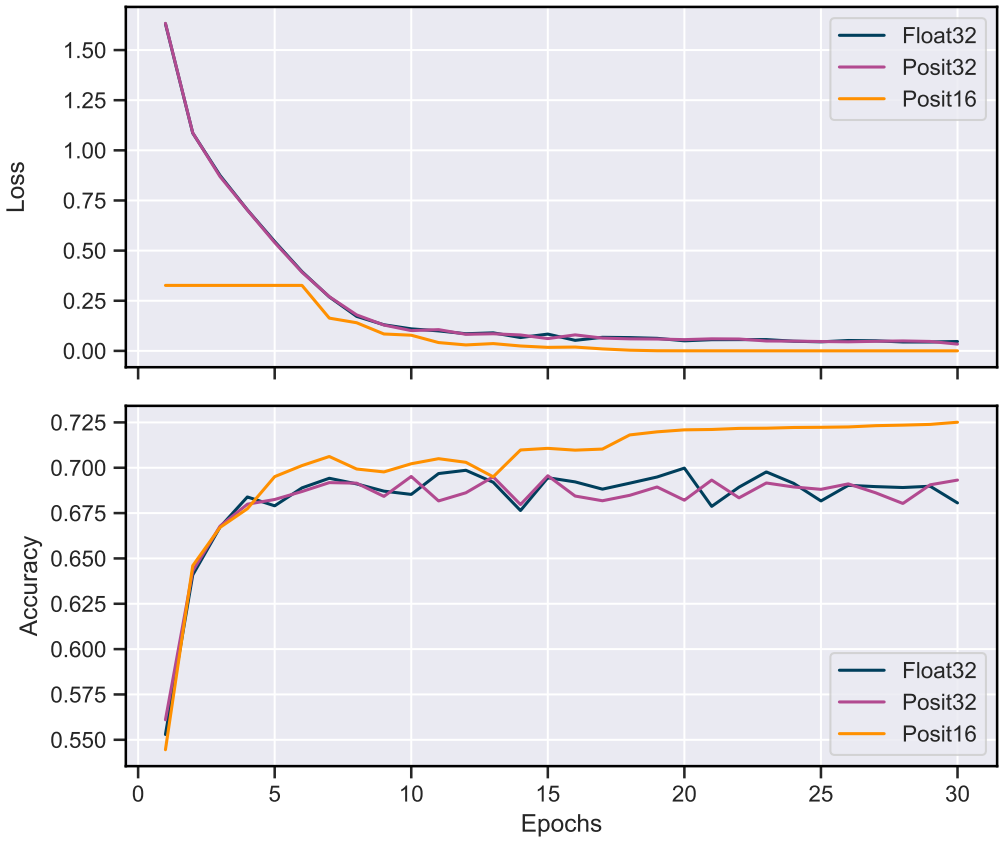


FIGURE 3.11: Learning process along CifarNet training on CIFAR-10

4 Design and Implementation of Posit Arithmetic Units[1]

One of the drawbacks hindering the development of applications in the posit format is the lack of native hardware support. Furthermore, when conducting experiments such as those in Section 3.4, the execution time is a highly limiting factor, since it increases tremendously when simulating all operations via software. The design of posit arithmetic units is therefore in the spotlight when comparing posit and float formats, and is the main objective of this Chapter, with a focus on arithmetic-related optimizations.

As explained in Section 3.1, DNNs mainly require multiplication and addition operations, and therefore, it makes sense to put more effort into designing such posit units. In fact, this is the minimum hardware required to perform DNN inference with posits, since other operations such as comparison (used for example in the ReLU function) can be performed as in the case of integers, thus using the logic of an ALU.

Section 4.1 reviews previous work regarding the design of posit arithmetic units. Section 4.2 presents the proposed designs for posit addition and multiplication algorithms, whose implementation results compared with the state of the art in Section 4.3.

4.1 Related work

Since the appearance of the PNS in 2017, the interest on a hardware implementation for this format has increased rapidly, and several approaches of hardware implementations for this arithmetic format have been proposed so far.

An open-source parameterized adder/subtractor was presented in [48], whose concepts where expanded in [49] to design a parameterized posit multiplier. These two works did not perform posit rounding, but fraction truncation, and used both a Leading One Detector (LOD) and a Leading Zero Detector (LZD) to determine regime value, which results in redundant area. Another parameterized design for posit arithmetic unit that included adders and multipliers and solved some of the shortcomings from previous works was presented in [50], where only a LZD was used at the cost of inverting negative regimes, and results were correctly rounded using the round to nearest even scheme. The same idea was applied in [51], where authors expanded their previous works [48], [49] and presented an open-source posit core generator which included a parameterized divider based on the Newton-Raphson method. Although that work seems totally parameterized, it does not

[1]The content of this chapter is originally published in the proceedings of 2020 IEEE International Symposium on Circuits and Systems (ISCAS 2020) [47]. The manuscript has been reformatted for inclusion in this MSc thesis.

support the configurations where $es = 0$. These configurations are important because allow to use more fraction bits, and are especially useful for low-precision arithmetic, and approximate computing approaches, as detailed in Section 2.4. The previous work [34] proposed the implementation of a generic posit multiplication algorithm into the open-source FloPoCo framework (version 4.1.2) [52], allowing to generate parameterized and combinational or pipelined operators in VHDL, and also including support for $es = 0$. This work extends the contents of [34] and proposes parameterizable algorithms for posit addition/subtraction and multiplication of two posit numbers, as well as algorithms for posit decoding and encoding, which are common stages for all the posit arithmetic operations. These algorithms are integrated into the FloPoCo framework, version 5.0. Hardware posit operators from aforementioned works have been written in HDL, but [53] presented a C++ template compliant with Intel OpenCL SDK that automatically generates and pipelines posit operations according to bitwidth and exponent size constraints.

It must be noted that since the posit standard [12] includes fused operations such as the fused dot product, and due to the importance of this operation for matrix calculus, some research and development for this kind of implementations has been done. Different fused multiply-add (FMA) units for posits are presented in [30], [54]–[57]. They make use of a quire register to accumulate the partial additions that are involved in the dot product, so the result is rounded only after the whole computation.

Finally, for the sake of completeness, recently there have been few efforts to integrate posit arithmetic with RISC-V cores [58]–[60]. RISC-V is an open standard instruction set architecture (ISA) that can be modified or extended with custom instructions for posit arithmetic.

As can be seen, posits are still in development. In terms of delay, area, and power, the posit arithmetic units are not yet competitive against their floating-point counterparts, and although they have shown some promising improvements in the field of NNs [17] there is still some debate about their real improvement [16].

4.2 Design of posit functional units

At hardware level, posits were designed to be "hardware friendly", i.e., to have similar (and even simpler) circuitry to existing floating-point units. The main encoding difference between float and posit formats is the fact that the second one includes the regime, which is a run-time varying scaling component. As will be shown in this Section, that is the main design challenge for posit units.

The computation flow for posit numbers, which is illustrated in Figure 4.1, is exactly the same as for floats. The fields of the operands must be decoded before performing the operation, and the result must be rounded and encoded into the same format.

4.2.1 Posit data extraction

Due to the variable-length fields of the posit format, the operand fields decoding flow slightly differs from the analogous stage for IEEE 754 numbers. The proposed posit decoding process (see Algorithm 2) first extracts the sign bit (MSB) and detects zero and NaR special cases by checking if remaining $n - 1$ bits are all 0. In case the input posit value is negative, the algorithm takes its absolute value via 2's complement, simplifying the data extraction process and future operations such as addition

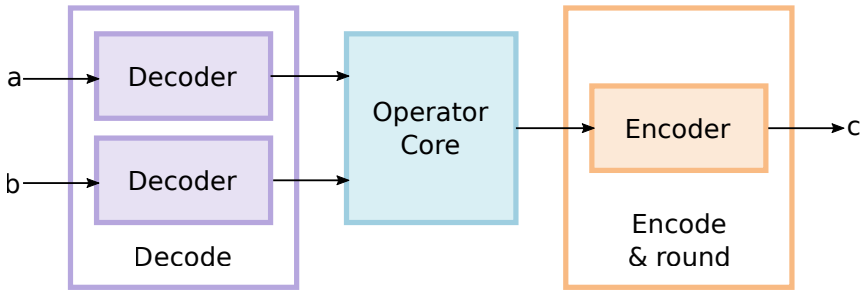


FIGURE 4.1: Generic computation flow for posits

of two posits. Then, the regime is computed and shifted-out and, afterwards, the exponent bits (if any) and fraction fields are effortlessly extracted. The main difference between the proposed posit decoder and previous implementations is the use of a Leading Zero or One Counter with integrated shifter (`LZOC+Shift`) instead of using separate detectors for decoding both positive and negative regimes [48], [49], [57] or using a single detector but always modifying the operand to deal with the same regime sign [34], [50], [51], [53] and then shifting out the regime. This approach simplifies the datapath substantially. Furthermore, the computed absolute value of the posit is not discarded, since it might be useful for operations such as the addition of two posits.

Algorithm 2 Posit data extraction

Require: $X \in \text{Posit}\langle n, es\rangle$
Ensure: $X = (-1)^{sign} * (2^{2^{es}})^{reg} * 2^{exp} * frac$
1: $sign \leftarrow X[n-1]$
2: $nzero \leftarrow \texttt{OR}(X[n-2:0])$ // Reduction OR
3: $z \leftarrow \neg(sign \vee nzero)$
4: $inf \leftarrow sign \wedge \neg(nzero)$
5: $twos \leftarrow (\{n-1\{sign\}\} \oplus in[n-2:0]) + sign$
6: $rc \leftarrow twos[n-2]$ // Regime sign
7: $shifted, count \leftarrow \texttt{LZOC+Shift}(twos, rc)$
8: $exp \leftarrow shifted[MSB-2 : MSB-es-1]$
9: $frac \leftarrow \{nzero, shifted[MSB-es-2 : MSB-es-frac_{size}-1]\}$
10: **if** $rc == 0$ **then**
11: $\quad reg \leftarrow -\{0, count\}$
12: **else**
13: $\quad reg \leftarrow \{0, count-1\}$
14: **end if**
15: **return** $sign, reg, exp, frac, z, inf, twos$

4.2.2 Posit data encoding and rounding

The pseudocode of posit output process is presented in Algorithm 3. Firstly, the scaling factor is split into exponent and regime. The latter is converted to positive to detect the overflow exception and aid the final regime encoding, which is done by right-shifting the exponent and fraction according to the sign of regime. Posits, same as IEEE 754 floats, follow round-to-nearest-even scheme. Therefore, the LSB,

Guard (G), Round (R) and Sticky (S) must be computed to perform a correct unbiased rounding. Finally, the result is obtained considering the final sign bit and special cases.

Algorithm 3 Posit data encoding

Require: $sign, sf, norm_f, inf, z$ fields of $R \in \text{Posit}\langle n, es\rangle$
Ensure: $R \in \text{Posit}\langle n, es\rangle$ and is correctly rounded
1: $exp_F \leftarrow sf[es-1:0]$
2: $reg_F \leftarrow sf[MSB-1:es]$
3: $rc \leftarrow sf[MSB-1]$
4: **if** $rc == 1$ **then**
5: $reg_F \leftarrow -reg_F$ // Get regime's absolute value
6: **end if**
7: **if** $reg_F > n-2$ **then**
8: $reg_F \leftarrow n-2$ // Regime overflow
9: **end if**
10: **if** $rc == 1$ **then**
11: $in_{shift} \leftarrow \{0, 1, exp_F, norm_f\}$
12: $offset \leftarrow reg_F - 1$
13: **else**
14: $in_{shift} \leftarrow \{1, 0, exp_F, norm_f\}$
15: $offset \leftarrow reg_F$
16: **end if**
17: $ans_{shf} \leftarrow in_{shift} \gg (offset, \neg rc)$
18: $ans_{tmp} \leftarrow ans_{shf}[MSB-1 : MSB-(n-1)]$
19: $LSB, G, R \leftarrow ans_{shf}[MSB-(n-1) : MSB-(n+1)]$
20: $S \leftarrow \texttt{OR}(ans_{shf}[MSB-(n+2):0])$
21: $round \leftarrow G \wedge (LSB \vee R \vee S)$
22: **if** $sign == 1$ **then**
23: $result \leftarrow \{1, -(ans_{tmp} + round)\}$
24: **else**
25: $result \leftarrow \{0, (ans_{tmp} + round)\}$
26: **end if**
27: **if** $inf == 1$ **then**
28: $R \leftarrow \{1, \{n-1\{0\}\}\}$
29: **else if** $z == 1$ **then**
30: $R \leftarrow \{n\{0\}\}$
31: **else**
32: $R \leftarrow result$
33: **end if**
34: **return** R

4.2.3 Posit adder/subtractor core

The addition of two posits is similar to how it is performed in the IEEE 754 format (see Algorithm 4). Firstly, the larger and smaller operands are detected. Note that this posit comparison can be done as integers [4]. The fraction of the smaller operand is right shifted (according to the difference of scaling factors of both operands) before being added/subtracted to the largest one. Then, the leading zeros from the resulting fraction are shifted out to get a normalized field. The final sign and scaling

factor are taken from the larger operand, subtracting the number of leading zeros from the fraction and adding one in case it is overflowed; therefore, the regime overflow may only occur when $es = 0$, and logic from encoding stage can be reduced under this consideration. Note that since operands are previously decoded, subtraction operation just differs on flipping a sign bit before performing the addition flow.

Algorithm 4 Posit addition

Require: $X, Y \in \text{Posit}\langle n, es\rangle$
Ensure: $R = X + Y$
1: $sign_X, k_X, e_X, f_X, inf_X, abs_X \leftarrow \texttt{Decoder}(X)$
2: $sign_Y, k_Y, e_Y, f_Y, inf_Y, abs_Y \leftarrow \texttt{Decoder}(Y)$
3: $OP \leftarrow sign_X \oplus sign_Y$
4: $inf \leftarrow inf_X \vee inf_Y$
5: $sf_X \leftarrow \{k_X, e_X\}$
6: $sf_Y \leftarrow \{k_Y, e_Y\}$
7: **if** $abs_X > abs_Y$ **then**
8: $\quad sign_L, sf_L, f_L \leftarrow sign_X, sf_X, f_X$
9: $\quad sf_S, f_S \leftarrow sf_Y, f_Y$
10: **else**
11: $\quad sign_L, sf_L, f_L \leftarrow sign_Y, sf_Y, f_Y$
12: $\quad sf_S, f_S \leftarrow sf_X, f_X$
13: **end if**
14: $offset \leftarrow |sf_X - sf_Y|$
15: $f_S \leftarrow f_S \gg offset$
16: **if** $OP == 1$ **then**
17: $\quad f_{add} \leftarrow f_L - f_S$
18: **else**
19: $\quad f_{add} \leftarrow f_L + f_S$
20: **end if**
21: $ovf_f \leftarrow f_{add}[MSB - 1]$
22: $norm_f, lzCount \leftarrow \texttt{LZC+Shift}(f_{add})$ // Normalize
23: $sf_{add} \leftarrow sf_L + ovf_f - lzCount$
24: $R \leftarrow \texttt{Encoder}(sign_L, sf_{add}, norm_f, inf)$
25: **return** R

4.2.4 Posit multiplier core

The posit multiplication process is detailed in Algorithm 5. It starts by detecting the output sign and corner cases. Computing the resulting fraction requires an integer multiplier whose inputs have a bitwidth of $frac_{size} = n - es - 2$ bits. The resulting scaling factor is obtained by adding both operand scales, plus the possible fraction overflow. In this case, this field has to be normalized shifting one bit to the right.

4.3 Evaluation of hardware implementation

The proposed designs have been implemented into FloPoCo, an open-source C++ framework for the generation of arithmetic datapaths that provides a command-line interface that inputs operator specifications and outputs synthesizable VHDL [52].

Algorithm 5 Posit multiplication

Require: $X, Y \in \text{Posit}\langle n, es\rangle$
Ensure: $R = X * Y$
1: $sign_X, k_X, e_X, f_X, z_X, inf_X \leftarrow \texttt{Decoder}(X)$
2: $sign_Y, k_Y, e_Y, f_Y, z_Y, inf_Y \leftarrow \texttt{Decoder}(Y)$
3: $sign \leftarrow sign_X \oplus sign_Y$
4: $z \leftarrow z_X \vee z_Y$
5: $inf \leftarrow inf_X \vee inf_Y$
6: $sf_X \leftarrow \{k_X, e_X\}$
7: $sf_Y \leftarrow \{k_Y, e_Y\}$
8: $f_{mult} \leftarrow f_X * f_Y$
9: $ovf_f \leftarrow f_{mult}[MSB - 1]$
10: **if** $ovf_f == 1$ **then**
11: $\quad norm_f \leftarrow \{0, f_{mult}\}$
12: **else**
13: $\quad norm_f \leftarrow \{f_{mult}, 0\}$
14: **end if**
15: $sf_{mult} \leftarrow sf_X + sf_Y + ovf_f$
16: $R \leftarrow \texttt{Encoder}(sign, sf_{mult}, norm_f, inf, z)$
17: **return** R

This tool allows operators to be automatically generated with the specified parameters and, therefore, to obtain posit operators for arbitrary values of $\langle n, es\rangle$ with the same base design. Figure 4.2 describes this VHDL generation flow[2]. The constructor method of each `Operator` (every datapath in the design inherits this class) places combinatorial VHDL code in the vhdl stream. It also builds up pipeline information, according to the specified frequency. Then the `outputVHDL()` method combines the VHDL stream and the pipeline information to form the VHDL code of the pipelined datapath. It also declares all the needed VHDL signals, entities, components, etc, so that a designer only has to focus on the architectural part of the VHDL code. The proposed addition and multiplication designs, which include support for correct rounding, has been made integrated in the official FloPoCo git repository, and the generated VHDL instances are publicly available to facilitate its use and dissemination. Simulations with extensive testing vectors are performed to verify the functionality of the proposed designs. These testing vectors have been generated with the SoftPosit reference library.

To evaluate the impact of proposed designs in terms of hardware resources, posit operators with standard configurations, this is $\langle 8, 0\rangle$, $\langle 16, 1\rangle$ and $\langle 32, 2\rangle$, have been generated with FloPoCo and synthesized using Synopsys Design Compiler with a 45 nm target-library and without placing any timing constraint. These operators have been compared with the state-of-the-art designs from Posit-HDL-Arithmetic [49][3] and PACoGen [51][4], which are open-source and publicly available under the BSD 3-Clause License. Unfortunately, those designs are limited to generate operators with $es > 0$, and no automatic pipeline is allowed. The former of these drawbacks is solved in the proposed designs, while FloPoCo, which can automatically generate pipelined operators for any given frequency, solves the latter. Thus, to have a fair comparison with previous works, combinational operators with $es = 2$ are

[2]Source [52].
[3]Source code accessed on January 10, 2021 from github.com/manish-kj/Posit-HDL-Arithmetic.
[4]Source code accessed on January 10, 2021 from github.com/manish-kj/PACoGen.

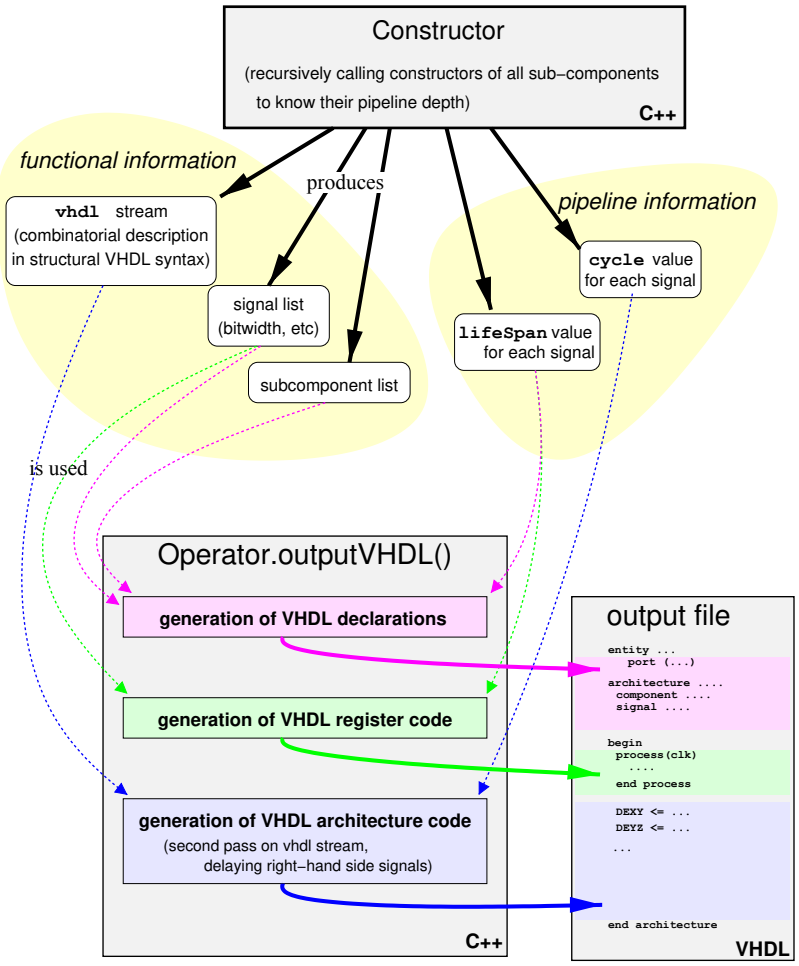


FIGURE 4.2: FloPoCo VHDL generation flow

generated (similar results are obtained for different *es* values). Besides this, floating-point addition and multiplication units have been generated for half (float16) and single (float32) precision with FloPoCo. However, it is important to mention that the floating-point designs provided by this tool do not include support for denormalized numbers or full exception handling, and thus, use less resources compared with a fully IEEE 754 compliant implementation. Nevertheless, this will be enough to have a reference between the posit operators and their floating-point analogs.

Standard cell synthesis results for area, delay, power and energy consumption of posit adders and multipliers are compared graphically in Figures 4.3 and 4.4, respectively. As can be seen, the proposed designs provide an overall reduction of area and power with respect previous implementations. With respect to the datapath delay of the operators, while the multiplier gets similar results as PACoGen (but worse than Posit-HDL), the proposed adder worsens in this respect compared to previous work. Despite this, energy consumption, which is the power–delay product, is in both cases lower for the proposed operators than for those from PACoGen, being very similar to that obtained with the Posit-HDL operators. However, it is important to recall that operators from Posit-HDL perform fraction truncation instead of proper rounding. This reduces logic (and therefore circuit area) in the encoding stage, as can be appreciated in Figures 4.3 and 4.4 when comparing operators from this generator against those from PACoGen. Finally, with respect to the floating-point operators, the results show that posit arithmetic units are still far from being

competitive in comparison with floating-point units. However, it is important to recall that while arithmetic units for floats have been optimized through decades of research, posits are still in development.

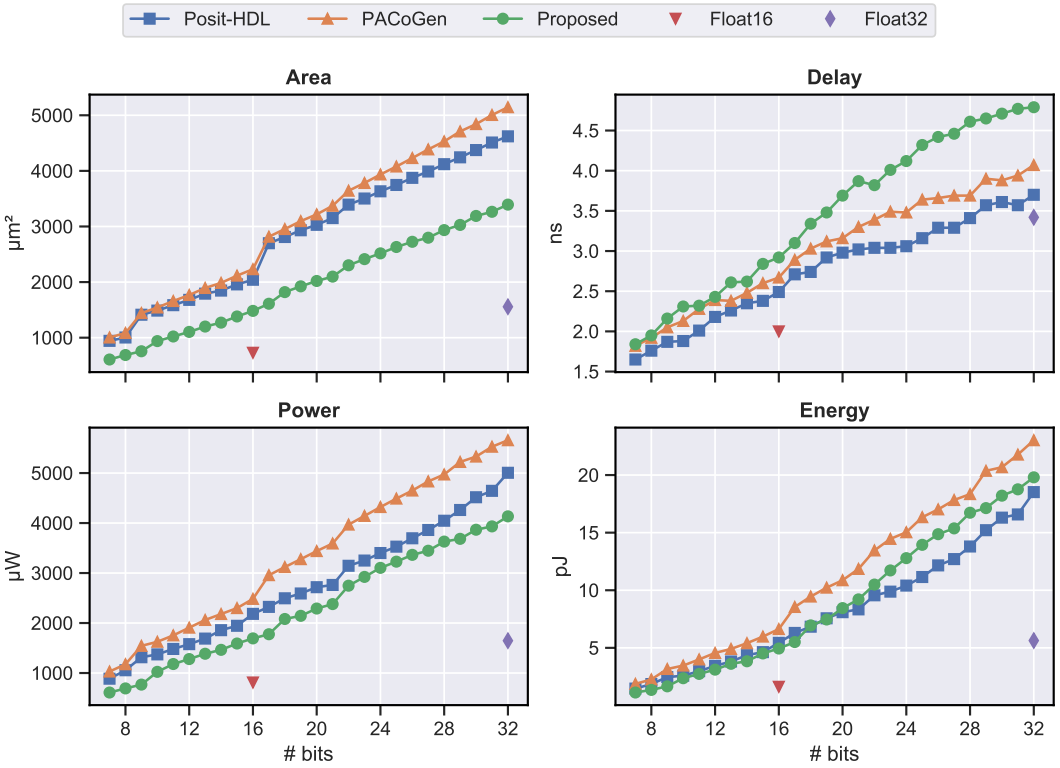


FIGURE 4.3: Posit$\langle n, 2\rangle$ and floating-point adder implementation results

For a more detailed comparison, Table 4.1 presents some metrics about the improvement of the proposed work with respect to the PACoGen core generator, which provides same results due to correct posit rounding. Positive values stand for reduction, while negative percentages indicate the augmentation of resources concerning the previous work. The improvements in terms of area and power savings are quite remarkable, especially for the adder unit, which archives reductions of more than 30% in average. Delay metrics perfectly reflect what was mentioned previously. In the case of multipliers, both designs obtain similar results (in fact, the mean improvement is 0%), while the proposed adder obtains higher delay that the baseline operator (as indicated by the negative reduction values).

TABLE 4.1: Hardware resource reduction of the proposed work with respect to PACoGen

	Posit Adder			Posit Multiplier		
Resource	Min	Max	Mean	Min	Max	Mean
Area	33.59%	47.59%	37.07%	21.12%	50.86%	32.76%
Delay	−24.93%	−1.09%	−12.59%	−9.0%	14.20%	0%
Power	26.92%	50.23%	32.71%	16.15%	56.80%	29.33%
Energy	8.80%	47.56%	23.37%	10.76%	59.93%	28.92%

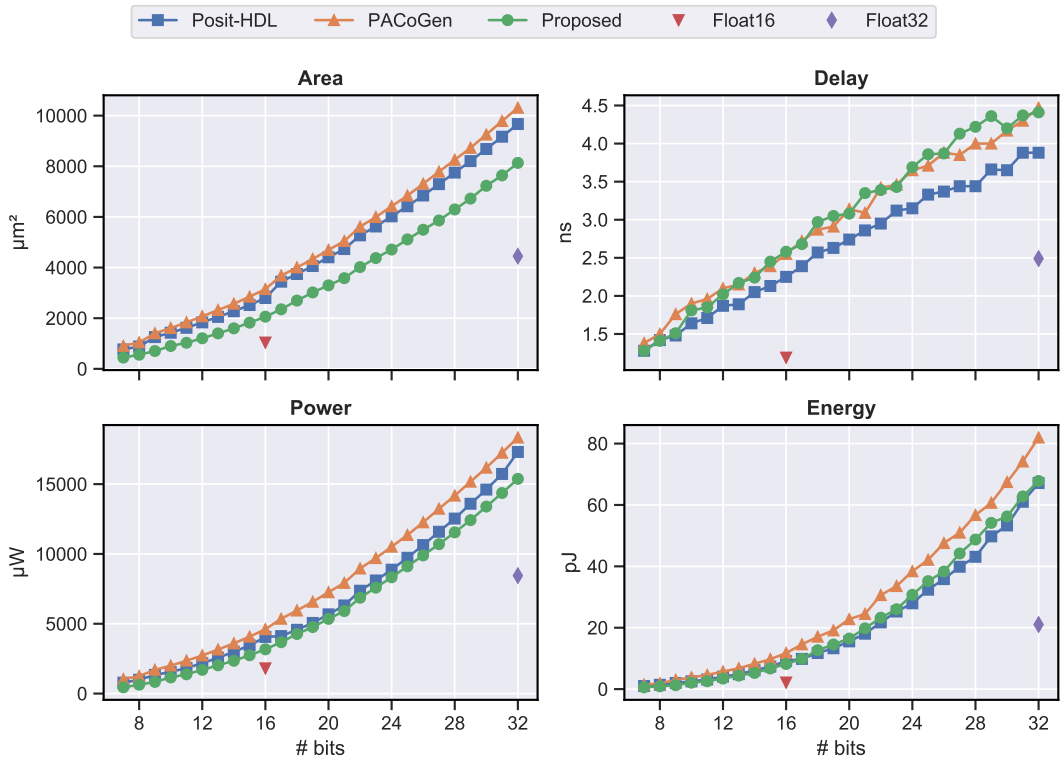


FIGURE 4.4: Posit$\langle n, 2\rangle$ and floating-point multiplier implementation results

5 | Conclusions and Future Work

5.1 Conclusions

For more than thirty years, the IEEE 754 has been the standard for floating-point arithmetic, used in all kinds of scientific applications in computer systems. Nonetheless, the recently introduced posit arithmetic is seen as a direct alternative to the now ubiquitous IEEE standard. Its use in the field of deep learning has been studied throughout this MSc thesis.

The recent advances on DNNs throughout the last decade have made them critical for many AI applications, including computer vision, speech recognition, and robotics, and providing even better performance than human being in many situations. However, while DNNs can deliver this outstanding accuracy, it comes at the cost of high computational complexity. In this area, posits are presented as ideal candidates to replace floats, providing the same precision with a smaller bitwidth. Chapter 3 presented Deep PeNSieve, a framework for performing both training and inference on DNNs entirely using the PNS. It allows training with Posit$\langle 32, 2\rangle$ and Posit$\langle 16, 1\rangle$, as well as performing post-quantization to Posit$\langle 8, 0\rangle$ with the support of the quire and the fused dot product. The major novelty of this work relies on performing the whole training with posits and evaluating this on CNNs, outperforming the state-of-the-art approaches, which only achieved training on smaller feedforward networks or with floating-point pre-training.

Due to the short life of posits, the design of arithmetic units for this format is still at an early stage of development. To date, few such designs have been proposed, even less open-source. Therefore, in Chapter 4 two parameterized algorithms for performing addition and multiplication of two posit numbers have been presented. Such algorithms have been integrated into the FloPoCo framework in order to obtain synthesizable VHDL code for any bitwidth (n) and exponent size (es) configuration. Results show a great improvement in terms of area, power and energy with respect to the state-of-the-art works.

All the presented work is open-source:

- Chapter 3: the deep learning framework based on the posit number system, Deep PeNSieve, is available at github.com/RaulMurillo/deep-pensieve
- Chapter 4: the parametric designs of posit operators are integrated in the `posit_utils` branch of the FloPoCo project (gitlab.inria.fr/fdupont/flopoco), and some VHDL instances can be found at github.com/RaulMurillo/Flo-Posit

5.2 Future work

Posit arithmetic is a relatively new topic in Computer Arithmetic, and there is still many different adaptations, tests, and experiments that can be done in the future to

get a deeper understanding of this novel format. In particular, this Master's thesis has been mainly focused on the use of posit arithmetic in DNNs, and the following ideas are proposed to continue with this research line:

- **Optimizations in the posit library.** As evidenced in Chapter 3, software emulation of posit datatype increases tremendously the execution time of applications, in comparison with floating-point format. In fact, the deep learning models trained in this work are not as big as the current state of the art models, such as MobileNet, VGG, GoogLeNet, ResNet, etc. [5]. Improving the proposed software library could reduce DNN training time, allowing experimentation with deeper DNN architectures and more complex datasets.

- **Mixed precision training.** The results obtained in this work show that posit-based DNNs can be trained not only with 32-bit posits, but with just 16 bits while obtaining similar accuracy as with standard single-precision floating-point. In addition, trained models can be quantized to Posit$\langle 8, 0 \rangle$ to perform inference (with the use of quire for GEMM operations) with insignificant accuracy degradation. Recent works [24] propose training DNNs using 8-bit floats for inference (with 32-bit accumulators for GEMM operations) and 16-bits floats for the backward weight update. This approach could be easily applied with posits, exploring the use of multiple precision and *es* values for the different training phases.

- **Use of approximate posit operators.** Machine learning, and in particular deep learning, are domains whose applications are error-tolerant. These kind of applications are a common target for approximate computing techniques, which reduce the accuracy of the results in pursuit of higher performance and less computing time [21]. As shown in Chapter 2, posits are able to approximate multiple functions (some of them frequently used in DNNs). The implementation of such functions, as well as other approximate units, such as multipliers, could be explored as a continuation of current work.

- **Designing a posit FMA operator.** Although the posit arithmetic units presented in Chapter 4 could be enough for deploying a posit-based DNN for inference into a hardware platform, such as an FPGA, it would be desirable to perform low precision inference, as well as training, on a hardware accelerator. For this purpose, the next step would be to design a posit unit for FMA. With this arithmetic unit, it would be possible not only to replicate the low precision inference experiments of this work, but to implement algorithms for more complex functions such as division and square root (using the Newton-Raphson method) that are used in the training stage of DNNs.

- **Development of a hardware accelerator for posit-based DNNs.** The posit arithmetic units, together with the DNN framework proposed in this work, could be integrated in a single platform that generates the necessary hardware and software interfaces to deploy posit-based DNNs on FPGAs. As recent works show [58]–[60], this hardware accelerator could be based on RISC-V cores.

Bibliography

[1] IEEE Computer Society, "IEEE Standard for Binary Floating-Point Arithmetic", *ANSI/IEEE Std 754-1985*, pp. 1–20, 1985. DOI: 10.1109/IEEESTD.1985.82928.

[2] IEEE Computer Society, "IEEE Standard for Floating-Point Arithmetic", *IEEE Std 754-2008 (Revision of IEEE 754-1985)*, vol. 2008, no. August, pp. 1–70, 2008. DOI: 10.1109/IEEESTD.2008.4610935.

[3] D. Goldberg, "What every computer scientist should know about floating-point arithmetic", *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991. DOI: 10.1145/103162.103163.

[4] J. L. Gustafson and I. Yonemoto, "Beating Floating Point at its Own Game: Posit Arithmetic", *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71–86, 2017. DOI: 10.14529/jsfi170206.

[5] S. Bianco, R. Cadene, L. Celona, and P. Napoletano, "Benchmark analysis of representative deep neural network architectures", *IEEE Access*, vol. 6, pp. 64 270–64 277, 2018. DOI: 10.1109/ACCESS.2018.2877890.

[6] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture", *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019. DOI: 10.1145/3282307.

[7] J. Dean, D. Patterson, and C. Young, "A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution", *IEEE Micro*, vol. 38, no. 2, pp. 21–29, 2018. DOI: 10.1109/MM.2018.112130030.

[8] D. Kalamkar *et al.*, "A Study of BFLOAT16 for Deep Learning Training", *arXiv e-prints*, pp. 1–10, 2019. arXiv: 1905.12322.

[9] J. L. Gustafson, *The End of Error*. Chapman and Hall/CRC, 2017. DOI: 10.1201/9781315161532.

[10] J. L. Gustafson, "A Radical Approach to Computation with Real Numbers", *Supercomputing Frontiers and Innovations*, vol. 3, no. 2, pp. 38–53, 2016. DOI: 10.14529/jsfi160203.

[11] A. A. Del Barrio, N. Bagherzadeh, and R. Hermida, "Ultra-low-power adder stage design for exascale floating point units", *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 3s, pp. 1–24, 2014. DOI: 10.1145/2567932.

[12] Posit Working Group, *Posit Standard Documentation*, 2018. [Online]. Available: https://posithub.org/docs/posit%7B%5C_%7Dstandard.pdf.

[13] U. Kulisch, *Computer arithmetic and validity: theory, implementation, and applications*. Walter de Gruyter, 2013, vol. 33. DOI: 10.1515/9783110301793.

[14] M. Cococcioni, F. Rossi, E. Ruffaldi, and S. Saponara, "Fast approximations of activation functions in deep neural networks when using posit arithmetic", *Sensors*, vol. 20, no. 5, p. 1515, 2020. DOI: 10.3390/s20051515.

[15] D. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)", in *4th International Conference on Learning Representations, ICLR*, 2016. arXiv: 1511.07289.

[16] F. de Dinechin, L. Forget, J.-M. Muller, and Y. Uguen, "Posits: the good, the bad and the ugly", in *Proceedings of the Conference for Next Generation Arithmetic 2019*, ACM, 2019, pp. 1–10. DOI: 10.1145/3316279.3316285.

[17] R. Murillo, A. A. Del Barrio, and G. Botella, "Deep PeNSieve: A deep learning framework based on the posit number system", *Digital Signal Processing: A Review Journal*, vol. 102, p. 102 762, 2020. DOI: 10.1016/j.dsp.2020.102762.

[18] V. Sze, Y. H. Chen, T. J. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey", *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017. DOI: 10.1109/JPROC.2017.2761740.

[19] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *nature*, vol. 521, no. 7553, pp. 436–444, 2015. DOI: 10.1038/nature14539.

[20] M. Cococcioni, E. Ruffaldi, and S. Saponara, "Exploiting Posit Arithmetic for Deep Neural Networks in Autonomous Driving Applications", in *2018 International Conference of Electrical and Electronic Technologies for Automotive, AUTOMOTIVE 2018*, 2018. DOI: 10.23919/EETA.2018.8493233.

[21] M. S. Kim, A. A. Del Barrio, H. Kim, and N. Bagherzadeh, "The effects of approximate multiplication on convolutional neural networks", *IEEE Transactions on Emerging Topics in Computing*, pp. 1–12, 2021. DOI: 10.1109/TETC.2021.3050989.

[22] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors", *nature*, vol. 323, no. 6088, pp. 533–536, 1986. DOI: 10.1038/323533a0.

[23] P. Micikevicius *et al.*, "Mixed Precision Training", *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 2017. arXiv: 1710.03740.

[24] N. Mellempudi, S. Srinivasan, D. Das, and B. Kaul, "Mixed Precision Training With 8-bit Floating Point", *arXiv e-prints*, 2019. arXiv: 1905.12334.

[25] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2323, 1998. DOI: 10.1109/5.726791.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks", *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017. DOI: 10.1145/3065386.

[27] J. Johnson, "Rethinking floating point for deep learning", *arXiv e-prints*, 2018. arXiv: 1811.01721.

[28] M. S. Kim *et al.*, "Efficient Mitchell's approximate log multipliers for convolutional neural networks", *IEEE Transactions on Computers*, vol. 68, no. 5, pp. 660–675, 2018. DOI: 10.1109/TC.2018.2880742.

[29] H. Kim, M. S. Kim, A. A. Del Barrio, and N. Bagherzadeh, "A cost-efficient iterative truncated logarithmic multiplication for convolutional neural networks", in *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, IEEE, 2019, pp. 108–111. DOI: 10.1109/ARITH.2019.00029.

[30] Z. Carmichael *et al.*, "Deep Positron: A Deep Neural Network Using the Posit Number System", in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2019, pp. 1421–1426. DOI: 10.23919/DATE.2019.8715262.

[31] Z. Carmichael *et al.*, "Performance-Efficiency Trade-off of Low-Precision Numerical Formats in Deep Neural Networks", in *Proceedings of the Conference for Next Generation Arithmetic 2019 - CoNGA'19*, ACM Press, 2019, pp. 1–9. DOI: 10.1145/3316279.3316282.

[32] H. F. Langroudi, Z. Carmichael, J. L. Gustafson, and D. Kudithipudi, "PositNN Framework: Tapered Precision Deep Learning Inference for the Edge", in *2019 IEEE Space Computing Conference (SCC)*, IEEE, 2019, pp. 53–59. DOI: 10.1109/SpaceComp.2019.00011.

[33] H. F. Langroudi, V. Karia, J. L. Gustafson, and D. Kudithipudi, "Adaptive Posit: Parameter aware numerical format for deep learning inference on the edge", in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, IEEE, 2020, pp. 3123–3131. DOI: 10.1109/CVPRW50498.2020.00371.

[34] R. Murillo Montero, A. A. Del Barrio, and G. Botella, "Template-Based Posit Multiplication for Training and Inferring in Neural Networks", *arXiv e-prints*, 2019. arXiv: 1907.04091.

[35] H. F. Langroudi, Z. Carmichael, and D. Kudithipudi, "Deep Learning Training on the Edge with Low-Precision Posits", *arXiv e-prints*, pp. 1474–1479, 2019. arXiv: 1907.13216.

[36] H. F. Langroudi, Z. Carmichael, D. Pastuch, and D. Kudithipudi, "Cheetah: Mixed Low-Precision Hardware & Software Co-Design Framework for DNNs on the Edge", *arXiv e-prints*, 2019. arXiv: 1908.02386.

[37] J. Lu *et al.*, "Evaluations on Deep Neural Networks Training Using Posit Number System", *IEEE Transactions on Computers*, vol. 14, no. 8, pp. 1–14, 2020. DOI: 10.1109/TC.2020.2985971.

[38] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning", in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*, USENIX Association, 2016, pp. 265–283. arXiv: 1605.08695.

[39] D. P. Kingma and J. Lei Ba, "Adam: A Method for Stochastic Optimization", Tech. Rep., 2014. arXiv: 1412.6980v9.

[40] B. Jacob *et al.*, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference", in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, IEEE Computer Society, 2018, pp. 2704–2713. DOI: 10.1109/CVPR.2018.00286.

[41] I. Hubara *et al.*, "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1", in *Advances in Neural Information Processing Systems*, vol. 29, 2016, pp. 4107–4115. arXiv: 1602.02830.

[42] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding", in *4th International Conference on Learning Representations, (ICLR)*, 2016. arXiv: 1510.00149.

[43] N. Wang *et al.*, "Training Deep Neural Networks with 8-bit Floating Point Numbers", *Advances in Neural Information Processing Systems*, pp. 7675–7684, 2018. arXiv: 1812.08011.

[44] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms", *arXiv e-prints*, 2017. arXiv: 1708.07747.

[45] Y. Netzer *et al.*, "Reading digits in natural images with unsupervised feature learning", pp. 1–9, 2011.

[46] A. Krizhevsky, "Learning multiple layers of features from tiny images", PhD thesis, University of Toronto, 2009.

[47] R. Murillo, A. A. Del Barrio, and G. Botella, "Customized Posit Adders and Multipliers using the FloPoCo Core Generator", in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2020, pp. 1–5. DOI: 10.1109/iscas45731.2020.9180771.

[48] M. K. Jaiswal and H. K. So, "Architecture Generator for Type-3 Unum Posit Adder/Subtractor", in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 2018-May, IEEE, 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351142.

[49] M. K. Jaiswal and H. K. So, "Universal number posit arithmetic generator on FPGA", in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, vol. 2018-Janua, IEEE, 2018, pp. 1159–1162. DOI: 10.23919/DATE.2018.8342187.

[50] R. Chaurasiya *et al.*, "Parameterized Posit Arithmetic Hardware Generator", in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, IEEE, 2018, pp. 334–341. DOI: 10.1109/ICCD.2018.00057.

[51] M. K. Jaiswal and H. K. So, "PACoGen: A Hardware Posit Arithmetic Core Generator", *IEEE Access*, vol. 7, pp. 74 586–74 601, 2019. DOI: 10.1109/ACCESS.2019.2920936.

[52] F. de Dinechin and B. Pasca, "Designing Custom Arithmetic Data Paths with FloPoCo", *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011. DOI: 10.1109/MDT.2011.44.

[53] A. Podobas and S. Matsuoka, "Hardware Implementation of POSITs and Their Application in FPGAs", in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2018, pp. 138–145. DOI: 10.1109/IPDPSW.2018.00029.

[54] J. Chen, Z. Al-Ars, and H. P. Hofstee, "A matrix-multiply unit for posits in reconfigurable logic leveraging (Open)CAPI", in *Proceedings of the Conference for Next Generation Arithmetic - CoNGA '18*, ACM Press, 2018, pp. 1–5. DOI: 10.1145/3190339.3190340.

[55] L. van Dam, J. Peltenburg, Z. Al-Ars, and H. P. Hofstee, "An Accelerator for Posit Arithmetic Targeting Posit Level 1 BLAS Routines and Pair-HMM", in *Proceedings of the Conference for Next Generation Arithmetic 2019 on - CoNGA'19*, ACM Press, 2019, pp. 1–10. DOI: 10.1145/3316279.3316284.

[56] Y. Uguen, L. Forget, and F. de Dinechin, "Evaluating the Hardware Cost of the Posit Number System", in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2019, pp. 106–113. DOI: 10.1109/FPL.2019.00026.

[57] H. Zhang, J. He, and S.-B. Ko, "Efficient Posit Multiply-Accumulate Unit Generator for Deep Learning Applications", in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 2019-May, IEEE, 2019, pp. 1–5. DOI: 10.1109/ISCAS.2019.8702349.

[58] S. Tiwari, N. Gala, C. Rebeiro, and V. Kamakoti, "PERI: A Posit Enabled RISC-V Core", *arXiv e-prints*, 2019. arXiv: 1908.01466.

[59] A. M. V. Sai, G. Bhairathi, and H. G. Hayatnagarkar, "PERC: Posit Enhanced Rocket Chip", in *CARRV at ISCA 2020*, 2020.

[60] R. Jain *et al.*, "CLARINET: A RISC-V Based Framework for Posit Arithmetic Empiricism", *arXiv e-prints*, vol. 1, no. 1, pp. 1–18, 2020. arXiv: 2006.00364.